数据科学与工程技术丛书

WILEY

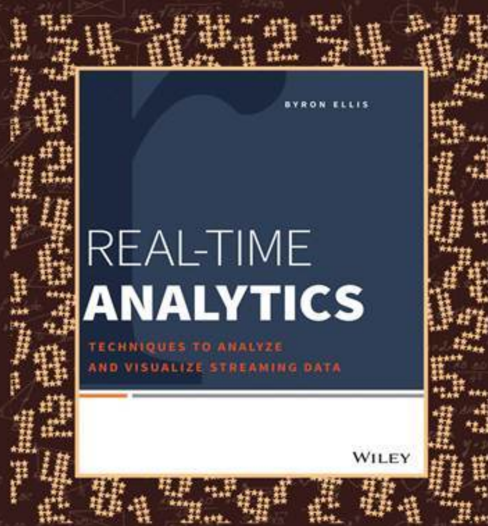
华章IT

实时分析

流数据的分析与可视化技术

[美] 拜伦·埃利斯（Byron Ellis）著

王晓伟 译



REAL-TIME ANALYTICS

TECHNIQUES TO ANALYZE AND VISUALIZE STREAMING DATA


机械工业出版社
China Machine Press

VISIT...

LANZAROTE Caliente.com

数据科学与工程技术丛书

实时分析：流数据的分析与可视化技术

Real-Time Analytics：Techniques to Analyze and Visualize Streaming Data

（美）埃利斯（Ellis，B.） 著

王晓伟 译

ISBN：978-7-111-53216-3

本书纸版由机械工业出版社于2016年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：＋ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

微信公众号 华章电子书（微信号：hzebook）

目录

译者序

前言

致谢

作者简介

技术编辑简介

第1章　流数据简介

1.1　流数据的来源

1.2　流数据的特别之处

1.3　基础架构和算法

1.4　总结

第一部分　流分析架构

第2章　实时流架构设计

2.1　实时架构的组件

2.2　实时架构的特性

2.3　实时编程语言

2.4　实时架构概览

2.5　总结

第3章　服务配置和协调

3.1　配置和协调系统的研发动机

3.2　维护分布式状态

3.3　Apache　ZooKeeper

3.4　总结

第4章　流分析中的数据流程管理
4.1　分布式数据流程
4.2　Apache　Kafka：高吞吐量分布式消息机制
4.3　Apache　Flume：分布式日志采集系统
4.4　总结
第5章　流数据的处理
5.1　分布式流数据处理
5.2　用Storm处理数据
5.3　用Samza处理数据
5.4　总结
第6章　流数据的存储
6.1　一致性哈希
6.2　“NoSQL”存储系统
6.3　其他存储技术
6.4　存储技术的选择
6.5　数据仓库
6.6　总结
第二部分　流分析与可视化
第7章　流度量的交付
7.1　流Web应用
7.2　数据可视化
7.3　移动流应用
7.4　总结
第8章　精确的聚集计算和交付

8.1 定时计数与求和

8.2 多分辨率时间序列的聚集计算

8.3 随机优化

8.4 时间序列数据的交付

8.5 总结

第9章 流数据的统计近似

9.1 数值计算库

9.2 概率和分布

9.3 参数估计

9.4 随机数产生器

9.5 抽样过程

9.6 总结

第10章 使用略图近似流数据

10.1 寄存器和哈希函数

10.2 集合

10.3 Bloom Filter

10.4 Distinct Value略图

10.5 Count-Min略图

10.6 其他应用

10.7 总结

第11章 流数据的应用

11.1 实时数据模型

11.2 用模型预测

11.3 监控

11.4　实时优化
11.5　总结

译者序

大概十年前，正是学术界对流数据的研究方兴未艾之时，我刚开始攻读博士学位。当时，数据库领域正致力于研究流数据管理系统（Data Stream Management System，DSMS），并发展出Stream、TelegraphCQ、Aurora等DSMS原型系统，数据挖掘领域则致力于开发各种经典数据挖掘算法的流数据版本。

然而，在工业界，Storm、Kafka、ZooKeeper等流分析基础架构的主要组件大多还处于内部开发的起步阶段，几乎不为人知。当时，除了Gigascope这个专用于网络监控的流数据管理系统，很少听说大规模实际应用流数据系统的案例。后来，在参与几个大规模网络安全监控项目的过程中，我也曾尝试应用学术界提出的几个DSMS来解决一些流分析问题，但效果均不理想。因此，从当时的情况来看，流数据的学术研究似乎离实际应用遥不可及，也正是由于这种悲观预期，我放弃了数据流这个研究方向，转而研究其他课题。

令人意想不到的是，近年来，随着Storm、Samza等流处理系统被贡献给开源社区，ZooKeeper、Kafka、Flume、Redis、MongoDB、Cassandra等相关系统也纷纷在流分析领域占据一席之地。短短几年间，一个完整的流分析基础架构似乎已经成型。在以Twitter、LinkedIn、Google为代表的大数据技术和应用的先行者中，这些系统已经各司其职，担负起艰巨的流分析任务。这样令人眼花缭乱的进步，怎能不让人感叹大数据浪潮的威力！

正是因为自己以前的研究经历，当看到本书目录时，我真的有眼前一亮的感觉。本书不仅详细讨论流数据的采集、传输、处理、交付等过程中涉及的各个主流系统，还介绍略图、Bloom Filter、抽样等重要数据结构和算法的基本原理，以及流分析中的应用。在流数据分析如火如荼的今天，这样一本具有很强实用性，同时又有一定理论深度的书实属不可多得。我相信，无论是对流分析感兴趣的入门级读者，还是在企业从事了多年流分析业务的研发人员和管理人员，都能从这本书中找到自己想要的东西。

最后，衷心感谢机械工业出版社对本书的翻译和出版所给予的大力支持，感谢编辑和审校老师付出的辛勤劳动。由于本人水平有限，翻译过程中难免有错误或疏漏。如果您发现翻译有误或者有其他疑问，欢迎您通过邮件批评指正。我的邮箱是gfkdwxw@aliyun.com.cn。

王晓伟

前言

概述及组织结构

流数据处理涉及软件开发和工程领域的许多不同问题。一方面，它需要一个灵活的基础架构，能够迅速便捷地移动数据；另一方面，处理速度要“跟得上”数据采集的速度，还要能扩展，以适应源源不断的数据流，由于这个限制，流数据处理很青睐从其他领域借鉴而来的数据结构。最后，一旦数据采集和处理完毕，应该利用数据做些什么？对于这一点，有一批可以直接在流数据上运行的应用，这些应用已经在大多数相关企业中发挥作用，还有更多的应用一直处于企业的考虑之中。本书将流数据的所有这些方面糅合在一起，既可以充当大众读者的入门书籍，又对更专业的技术人员有参考价值。我们希望，通过阅读本书，你能够建立足够的自信，在企业中从头到尾实施一个流数据的概念验证项目，并尝试将其应用到生产环境。为了实现以上目标，不仅需要建立基础架构，还需要实现相关算法，为此我们将本书划分为两个独立的部分。

第一部分介绍流数据系统本身的基础架构，以及系统的运营问题。如果数据是流式的，但是仍然以批量的方式处理，那就不再是流数据，这些只是碰巧连续采集得到的批量数据，这样的数据对许多用例完全够用。但是，本书的假设是，人们已经认识到，如果数据生成后不久就可以由最终用户使用，这样会大有裨益。鉴于此，本书涵盖了完成这种任务所需要的工具和技术。

我们首先介绍流框架底层的概念和特性，包括流处理系统的各种组件。并不是所有项目在起始阶段都会用到这些组件，但所有成熟的流基础架构最终都会用到它们。然后我们在流基础架构的关键特性，即可用性、可扩展性和延迟的背景下讨论这些组件。

第一部分剩下的内容聚焦于实现或配置每个组件的具体细节。由于组件框架的广泛应用，多半已经不需要编写多少代码来实现组件。取而代之的是安装、配置，或者定制工作。

第3章和第4章介绍构建和协调数据移动系统所需要的工具。根据所处环境的不同，可能需要开发软件来直接与该系统集成，或者修改已有软件使之适应该系统。我们将讨论这两种途径的利弊。

数据一旦被移动，就必须得到处理，并最终存储起来。这是第5章和第6章的内容。这两章介绍主流的流处理软件，以及对数据存储系统的选择。

第二部分用流基础架构解决各种问题。仪表板与预警系统是流数据采集最早的应用，也是第二部分介绍的第一个应用。

第7章讨论从流环境向最终用户的数据交付问题。这是构建仪表板以及其他监控应用所使用的核心机制。一旦交付，数据就必须呈现给用户，因此，该章还包括一节内容，专门讨论如何在基于Web的环境中建立仪表板可视化。

当然，所交付的数据经常要由处理系统进行聚集计算。第8章涵盖流环境下的数据聚集计算问题，特别是对多分辨率时间序列数据的聚集计算

问题，该结果数据最终会被交付给第7章讨论的仪表板应用。

对数据进行聚集计算之后，数据中有什么样的模式这个问题就浮出了水面。是否有随时间变化的趋势？某个行为是否与先前观察到的行为有明显不同？要回答这些问题，需要一些统计学和随机过程方面的知识（一般来讲，这两方面的知识可以解答关于大规模数据采集的任何问题）。第9章对统计学和概率论的基础知识进行了简要介绍。其中还介绍了统计抽样的概念，它可以用来计算比简单的聚集更加复杂的度量。

抽样是对复杂度量进行近似的传统机制，但我们还可以通过其他机制更好地计算某些度量，略图（sketch）概率数据结构就是这样的一种机制。第10章将讨论略图，这需要大量使用第9章的概率论知识。略图通常具有更快的更新速度并占用更少的内存，因此特别适合流环境。

最后，第11章讨论聚集计算之外能够应用于流数据的一些更深入的话题。篇幅所限，这一章仅对所涉及的众多话题作简要介绍，实际上每个都可以单独成书。第一个话题是来自统计学和机器学习领域的流数据模型。这些模型是预测等许多应用的基础，用来估计未来的数据值。由于数据是流式的，可以将这些预测值与实际值相比较，据此再对模型进行修正。预测的应用很广泛，其中包括第11章中讨论的异常检测。

第11章还简要介绍优化和A/B测试。如果你可以预测用户对两个不同网站设计的反应，就可以利用这些信息，向用户展示具有较好预期反应的网站设计。当然，预测并不是完美的，也有可能对各个网站的设计效果给出糟糕的估计。要改进对特定网站设计的预测，唯一途径是收集更多的数据。这样，你就需要确定各网站设计要多久展示一次，从而保证在改进预

测的同时，不会由于展示了非主流的设计，而牺牲展示效果。第11章给出一种常用于解决这类问题的简单机制，称为多臂赌博机（multi-armed bandit）。利用这种机制，你可以站在一个比较高的起点上对优化问题进行更深入探索。

本书面向的读者

我们曾在开篇提到过，本书意在吸引从事软件行业的广大读者，针对的是有兴趣开始涉足流数据及其管理的技术人员。正因如此，我们希望读者按顺序阅读，最终可以全面掌握流数据分析的基础知识。

即便如此，对于只熟悉这个领域的一部分内容，而对其他部分不甚了解的专家来说，本书也是值得一看的。例如，数据分析师或数据科学家可能在第9章和第11章的统计学方法方面有很强的专业背景，也可能对第7章的仪表板应用以及第8章的聚集计算技术有一定的经验，但对第10章的概率数据结构可能没有太多了解。如果不是要实际实现基础架构，而是想理解对设计的权衡是怎样影响所分析的数据的交付的，他们也可能会对前六章感兴趣。

与之类似，关注运营和基础架构方面的读者可能对第1～6章所讨论的话题颇有了解。他们可能没有接触过特定的软件，但肯定处理过许多类似的问题。本书第二部分，即第7～11章可能对他们更有吸引力。系统监控是流数据的首批应用之一。异常检测这样的工具可以在诸如健壮的错误检测机制这样的开发中派上用场。

需要的工具

不管你喜欢与否，数据基础架构领域大多以Java虚拟机为基础。这里的原因比较复杂，但不管怎样，它始终是本书所需要的工具。本书中使用的软件和示例是基于Java 7开发的，通常也可以支持Java 6或Java 8。读者应确保操作系统安装了合适的Java开发工具包。

由于要使用Java，有必要安装一个编辑器。本书中的软件是用Eclipse编写的，项目也是用Maven构建系统来组织的。安装这两个软件将帮助你构建本书包含的示例。

本书也使用了其他软件包，具体安装方法会在相应的章节中讲到。

书中使用了一些基础的数学术语和公式。如果你的数学知识有些生疏，觉得这些概念有点难，Sheldon Ross的《A First Course in Probability》会对你有所帮助。

配套网站

本书配套的网站中包含了每一章所有例子的代码包。每章的代码都划分成独立的模块。

有的代码在多章中共用。这种情况下，代码被复制到每个模块，以保证这些模块是自包含的。

网站还包含Samza源代码的拷贝。Samza是一个新兴的项目，其代码

库更新很快，这导致在编写和编辑相关章节的时候，本书中的示例可能已经难以运行。为了避免对读者造成困扰，我们在网上放了一个能与本书中代码良好兼容的项目版本。请参考http://www.wiley.com/go/realtimeanalyticsstreamingdata 。

扬帆起航

闲话少说，是时候踏上实际构建流数据系统的精彩旅程了。本书所讲的技术虽不是流数据处理的独门秘籍，但却都是我付出多年心血摸索出来的经验。我个人认为这些技术都非常好用，虽然在这个激动人心的时代，新事物始终会层出不穷。不管怎样，我希望，通过阅读这本书，你至少能少走弯路。

去找一些数据，让我们扬帆起航。

致谢

在写这本书之前，每当在致谢部分看到“有太多要感谢的人”这句话，我都觉得是陈词滥调。现在我才明白，这样的话真的是发自肺腑。对于这本书，我确实有太多需要感谢的人，实在难以在这里对他们一一列举。

不过，我还是要为其中一些人的贡献专门致谢，尽管他们有的人未必知道这本书。第一个当然是Wiley出版社的Robert Elliot，他欣赏我的演讲并认为可以写成一本近400页的书。没有他，就没有本书的面世。我还要感谢Justin Langseth，他是我那次演讲的合作者，很遗憾他没有能与我合作编写本书。希望我们还有机会再次合作。还应该感谢由Kelly Talbot领导的编辑，Charlotte、Rick、Jose、Luke和Ben，是他们帮助我找到和纠正了许多错误，使项目始终按计划推进。如果书中还有什么错误，那绝对是我的问题。

我要感谢DDG所有的常客，本书中的软件至少有一半，甚至可能超过80%，都是我端着啤酒杯聊天得来的。对于一个非正式的松散聚会，这真的很值得。感谢Mike第一个邀请我一同前往，感谢Matt和Zack这些年来举办这么多活动。

最后，我要感谢我这些年来的同事们。应该为你们忍受我各种轻率计划和笨拙补救颁发奖牌。特别感谢adBrite的全体成员，我们一起做出了许多很酷的东西，这些东西仍然处于领域的前沿。感谢Caroline Moon，允许分析人员使用新奇的“Hadoop”，以及收集更多的服务器。特别感谢Daniel Issen和Vadim Geshel。我们并不是经常看法一致（可能仍然无法

做到），但本书的内容很多都来源于我与他们两人的争论。

作者简介

Byron Ellis 是Spongecell公司的CTO，该公司是一个总部位于纽约的广告技术公司，在旧金山、芝加哥和伦敦设有办事处。他负责公司的研发和计算基础设施的维护工作，在加盟Spongecell之前，他是在线交互技术“领头羊”企业Liveperson公司的首席数据科学家。他还在当时世界最大的广告交换公司之一adBrite担任过多项职务。他拥有哈佛大学统计学博士学位，攻读博士学位期间主要研究高吞吐量生物学实验数据中网络结构的学习方法。

技术编辑简介

Jose Quinteiro 有20年技术经验，参与过许多终端用户、企业、Web软件系统和应用的设计与开发工作。他对于包括前后端的设计和实现在内的全套Web技术有着丰富经验。Jose在威廉玛丽学院获得化学学士学位。

Luke Hornof 拥有计算机科学博士学位，曾参与创建了多个成功的高科技初创企业。他在编程语言方面发表了十多篇同行评审的论文，曾为微处理器、广告和音乐行业开发过商用软件。他目前的兴趣之一是使用数据分析技术来改善Web和移动应用。

Ben Peirce 在Spongecell广告技术公司负责研究工作和基础设施的管理。加盟Spongecell之前，他在医疗健康技术初创企业担任过多项职务，他还是SET Media公司的联合创始人之一，该公司是一个视频广告技术公司。他在哈佛大学工程与应用科学学院获得博士学位，研究方向是控制系统和机器人。

第1章　流数据简介

当今世界每天都在以更快的速度发展。人和地区之间的联系越来越紧密，人员和企业的反应速度也越来越快。这越来越接近于人类反应能力的极限，因此人们创建了一些工具处理决策者面临的海量数据，分析、展现这些数据，并在事件发生的时候及时做出反应。

这些数据的采集和处理有许多应用领域，其中某些领域将在下一节介绍。我们将在本章后面讨论这些应用。这些应用需要一个专门针对流数据的基础架构和分析方法。幸运的是，与以前的批处理类似，流基础架构的最新技术关注于使用商用硬件和软件来构建系统，而不是像互联网时代之前那样使用专用的实时分析系统。这样的系统与基于云的灵活环境结合在一起，使得实时系统的实现对几乎所有企业都变得触手可及。利用这些商用系统，企业能够实时分析数据，并且能随着时间的推移对基础架构加以扩展，以满足企业未来发展和变革的需求。

本书面向广泛的大众读者以及企业的技术实现人员，目的是使他们能够轻松应对全栈应用。当实时项目进入某个阶段时，它们应该成为易于修改的、敏捷且自适应的系统，这需要用户除了了解所关注的领域，还要对全栈应用有清晰的整体理解。“实时”不仅针对数据本身，对于新的分析任务的开发也同样适用。许多原来规划良好的项目之所以失败了，是因为项目的实现耗时太久，导致项目发起人转而从事其他项目，或者干脆忘记了当初为什么需要这些数据。通过让项目变得敏捷和可递增，就可以尽可能地避免这种情况。

本章划分为三小节，分别对应三部分内容。第一节介绍一些常见的流数据来源和应用。这部分内容基本上按照时间先后排序，并对流数据基础架构的起源给出了一些背景介绍。这个介绍不仅具有历史意义，更重要的是，其中介绍了当初开发的许多工具和框架，用来解决这些领域中的问题，其设计反映了它们当时所处领域的一些独特挑战。例如，第4章中介绍的数据移动工具Kafka就被开发作为Web应用工具，而第5章中涉及的处理框架Storm，起初就是由Twitter开发，用于处理社交媒体数据的。

第二节涵盖流数据三个重要特性：持续数据交付、松散结构数据和高基数（high-cardinality）数据集。第一个特性当然是将系统定义为实时流数据环境的首要特性。其他两个特性尽管并不是流数据所特有的，却为流数据应用的设计者带来了独特的挑战。这三者一起构成了本质意义上的流数据环境。

第三节简要介绍对流数据使用基础架构和算法的重要性。

1.1　流数据的来源

流数据的来源多种多样，这一节只介绍一些主要的数据类别。尽管出现了越来越多的数据来源，也会有许多专用的数据来源，本节还是只讨论令流数据受到关注的应用领域中的数据类别。我们主要按照时间先后对这些应用领域排序。对书中讨论的软件追根溯源，它们中很多都是在解决这些特定应用领域中的问题时产生的。

书中给出的数据移动系统，刚开始是为LinkedIn、Yahoo！和Facebook的网站分析与在线广告处理数据。设计这样的处理系统是为了应对Twitter和LinkedIn这样的社交网络所带来的社交媒体数据处理的挑战。

Google公司的商业帝国与在线广告息息相关，它们大量使用的高级算法与第11章中的算法异曲同工。Google对一项名为深度学习的技术特别感兴趣，该技术利用超大规模神经网络来学习复杂模式。

通过使物联网以及其他高度分布式的数据采集手段经济可行，这些系统甚至正在开辟数据采集和分析的全新领域。我们希望，通过对以往应用领域的勾勒，能够对这些技术尚未预见的应用有所启示。

1.1.1　运行监控

实体系统的运行监控是流数据最初的应用。刚开始是用专用硬件和软件来实现的（在计算机时代之前，甚至是用模拟和机械系统）。当今最常见的运行监控的用例就是对为互联网提供动力的实体系统进行性能监视。

这些数据中心有几千个甚至数万个独立的计算机系统。这些系统持续不断地记录自己的物理状态数据，包括处理器温度、散热风扇的转速，以及消耗的电量。它们还记录磁盘驱动器状态信息和基础的运维度量，如处理器负载、网络活动，以及存储访问时间。

为了对所有这些实体系统进行监控，并及时发现问题，数据通过各种机制实时采集和聚集。刚开始的系统往往是专门的特殊机制，但是当这类技术开始应用于其他领域时，这些系统就开始使用与其他数据采集机制相同的采集系统了。

1.1.2 Web分析

通过电子商务和在线广告引入的商用化Web，引发了对网站行为的监控需求。与报纸的发行量类似，一天之中某个网站的不同访问者数量是重要的信息。对电子商务网站而言，与网站访问的人数相比，更重要的数据是人们浏览的各种商品以及这些商品之间的相互关系。

为了分析这些数据，许多专门的日志处理工具被研发出来，并推向市场。随着大数据以及Hadoop等工具的兴起，许多Web分析的基础架构转变为这类大规模批处理系统，用来实现推荐系统以及其他分析任务。显而易见，还可以通过对网站的结构进行实验，来看它们如何影响我们感兴趣的各种度量。这称为A/B测试，因为它采用与验光师确定最好搭配方案时相同的方法，即让两个选择相互对比来确定哪一个最好。这些测试多数是顺序进行的，但这存在许多问题，其中最重要的问题是进行对比研究所需要的时间量。

随着越来越多的企业挖掘其网站数据，对缩减反馈环节的时间，以及以持续不断的方式采集数据的需求愈发重要。使用系统监控领域的工具，使得实时采集数据和执行A/B测试这样的任务得以并行，而不再需要顺序执行。随着度量维数以及对适当审计的需求（缘于用于计费的度量）的增加，数据分析领域开发了本书提到的许多流基础架构，将数据从分散在世界各地的网站服务器安全地移动到处理和计费系统中。

尽管这类数据通常位于企业内部而不公开，它们仍然是广泛的海量信息来源。其应用范围从用于计费的简单聚集，到用于根据当前浏览历史（在Netflix公司则是影片观看历史）对产品推荐的实时优化。

1.1.3　在线广告

实时数据的主要用户和主要来源之一就是在线广告。在线广告的最初形式与对应的印刷广告类似，都是提前几个月“购买”。随着广告交换和实时竞价基础架构的兴起，广告市场中很大一部分流量更具流动性，并且这部分流量始终在不断增长。

对于这些应用来说，在不同环境和不同网站上所花费的资金是以按分钟计费的方式管理的，这与股票市场几无二致。另外，这些广告购买需求经常以某种度量进行管理，例如购买数量（称为转化（conversion））甚至是某个广告的点击率这样更简单的度量。当访问者通过现代的广告交换进入网站，就会有许多竞价代理（每次可能有30或40个）被调用，它们实时地对页面点击量进行竞价。竞拍过程结束后，竞价获胜一方的广告就会被显示出来。这通常在加载页面其他部分的时候发生，所花费的时间一般

不超过100毫秒。如果页面包含多个广告（这很常见），经常会对所有这些广告进行竞拍，有时会涉及多个不同的广告交换。

这个过程中所有的参与方，包括广告交换、竞价代理、广告客户和广告发行商，都会为了各自不同的目的实时采集数据。对于广告交换来说，数据是竞价过程的关键部分，同时对于实时反馈机制也很重要，这种机制用途广泛。例如对欺诈流量交易的监控，以及限制对各参与方广告显示的访问等其他风险管理活动。

广告交换双方的广告客户、广告发行商和竞价代理也都实时采集数据。他们的目标是管理和优化当前运行的竞价活动。从竞价的选择（如果是广告客户）或者“保留”价格（广告发行商），到决定对某个特定类型的流量哪个广告交换的价格最好，数据都是以分秒必争的方式管理的。

一个大型的广告活动或大规模网站可以轻松拥有几千万甚至几亿的页面点击量。如果将点击和转化等其他事件包括在内，事件的数量可以轻易翻番。一个竞价代理通常一次性代表许多不同的广告客户或广告发行商，因此每天采集到几亿到几十亿量级事件的情况很常见。即使是中等规模的广告交换，中间方每天也可以有几十亿的事件。所有这些数据一旦生成，都会被采集、移动、分析和存储。

1.1.4　社交媒体

另一种新近的海量数据来源是社交媒体，特别是像Twitter这样的开放数据来源。截至2013年年中，Twitter声称每天能采集到50亿条推文，每

秒15万条左右。这个数字当然还会不断上升。

这些数据被实时采集和传播，成为新闻媒体和全世界其他用户的重要信息来源。在2011年，纽约的Twitter用户收到华盛顿特区传出的发生地震的消息，大约30秒后地震才波及纽约。

加上Facebook、Foursquare，以及不断涌现的各种通信平台等其他数据来源，社交媒体数据就显得极度庞大和多样。来自Web分析和在线广告这类应用的数据尽管具有很高的维度，但一般具有良好的结构，每一维上的数据（例如投资金额或地理位置）都是很好理解的定量数值。

然而，社交媒体数据通常是高度非结构化的，至少数据分析人员是这样理解的。这类数据通常是“自然语言”形式，必须由自动化的系统来解析、处理并以某种方式理解。这使得社交媒体数据尽管极为丰富，同时作为实时数据来源，其处理又极具挑战性。

1.1.5　移动数据和物联网

2007年苹果公司发布了iPhone，一个最激动人心的新数据来源也随之横空出世。虽然可以存储蜂窝数据的便携式计算机已经存在至少十年之久，类似于黑莓手机这样的设备也使商业用户可以在手掌上处理数据，但是这些设备本质上仍然是专门工具，其管理也是专门化的。

IPhone、Android手机以及其后的其他智能手机，使得蜂窝数据成为消费技术，用户数量激增所带来的规模经济也就随之而来。智能手机还使得我们口袋里都装上了一个通用计算机。智能手机不仅能够向在线服务反

馈数据，还能使用低功耗蓝牙这样的技术与附近的设备通信。

得益于这种新的基础架构，类似于“可穿戴计算机”这样的技术，使得我们可以用与以往二十年间测量虚拟世界同样的方式测量现实世界。这类技术的应用有的荒唐可笑，有的则很有用，有的甚至有点吓人。试想一下，当用户晚上睡眠不好，第二天需要提神时，测量睡眠活动的手环可以触发自动咖啡壶。冲泡咖啡的味道甚至可以用作起床闹钟。这些系统间的通信不必像过去50年间各种“智能家居”演示项目所设想的那样直接或专门化。今天，使用If This Then That（IFTTF）这样的工具，以及建立在与本书相似的基础架构之上的公开可得的系统就可以实现这些工具。

言归正传，我们可以通过远程设备实时测量重要的生物特征数据。以前这只有使用高度专门化的昂贵设备才做得到，因此限制了这类数据只能在太空探索这样高度紧张的环境中应用。现在可以长时间采集个人的生物特征数据（在统计学里这称为纵向数据（longitudinal data）），然后将其与其他用户的数据合并在一起，从而得到人类生物特征识别标准的更加完整的图谱。我们不再需要每年一次，让病人在冰冷的房间穿着纸质衣服测量血压，而是可以在一段时间内持续跟踪他的血压，从而做到防患于未然。

除了医疗卫生领域，“智能微尘”的概念也由来已久，“智能微尘”是指通过对散布在世界的某个区域的数量众多的廉价传感器进行远程查询，来采集有意义的数据。这些设备的局限性主要是，制造这些相对专门的设备所耗费的成本。随着数据采集硬件和软件（如智能手机）的商业化，这个问题得到了解决，物联网也就应运而生。使用这类技术，不仅人们可以对

自己进行持续监控，物体也可以对自身进行持续监控。这有着各种各样的潜在应用，例如城市交通管理，以及通过更好地监控土壤条件来提高农业生产效率。

上述讨论的重点在于，现在信息可以通过商业化系统来流动，而不是像以前那样通过专门的数据采集硬件和软件。这些商业化系统都是可以买到的，分析数据所需要的软件也是现成的。需要开发的只有采集数据的新型应用而已。

1.2　流数据的特别之处

流数据与其他类型的数据区别包括多个方面。本节涵盖了最重要的三个方面，分别是数据的“始终在线”属性、松散且不断变化的数据结构和高基数带来的挑战。在对本书中给出的各种流框架设计和实现的决策中，所有这三方面都起到重要作用。流数据的这些特性尤其会影响第5章中介绍的数据处理框架，在数据移动工具的设计决策上也会有所体现，即对于经过它们的系统的信息，有意不强加数据格式，从而使灵活性最大化。本节剩下的内容会对这三个方面做更深入的介绍，为你进入第2章（介绍流架构的组件和需求）提供一些背景知识。

1.2.1　始终在线，持续流动

第一个方面有些显而易见：数据流是流式的。数据始终可得，新数据不断生成。这对于任何采集和分析系统的设计都有些许影响。首先，采集系统本身要有非常好的健壮性。主要的采集系统一旦宕机，就意味着数据永久丢失。在设计边缘采集器时，这一条非常重要，你应该牢记于心。第2章会对此做更详细的讨论。

其次，数据是持续流动的这一事实意味着系统的处理要能够跟得上数据的流动速度。如果需要2分钟才能处理1分钟的数据，系统就无法长时间保持实时。最终情况会糟糕到不得不丢弃一些数据让系统的处理跟上来。在实际情况中，系统仅仅能实时地“跟上”数据是不够的。因为无论是按计划停机这样的人为原因，还是网络中断这样的灾难性故障，系统都会因此

全部或者部分宕机。

如果没有为这种无法避免的情况提前计划，导致系统的处理速度只能与事件产生速度相同，这就意味着系统宕机时，采集器存储的数据会导致系统产生延迟。换句话说，能够在1分钟内处理1小时数据的系统，不需要太多干预就能够很快跟上。一个有着很好的水平可扩展性的成熟环境（第2章中也会讨论这个概念）甚至能够实现自动扩展。在这种情形下，随着延迟的增加，会临时加入更多的处理能力，从而使延迟回到可以接受的限度之内。

从算法的角度来看，流数据的持续流动特性是一把双刃剑。好的一面是，很少会出现数据不足的情况。如果我们的分析需要更多数据，等待足够的数据到来就可以了。可能需要长时间的等待，但同时可以进行其他的分析任务，从而提前预知如何继续进行后续分析。

不好的一面是，过去80年间所开发的统计工具多数集中于离散实验。许多标准的分析方法并不一定很适合流式数据。例如，在流环境下使用时，“统计显著性”这个概念就变得有点奇怪。许多人将其视为数据采集的某种“停止规则”，但实际上并非如此。用来做显著性检验的p-value本身是随机值，可能低于临界值（通常是0.05），然而观察到的下一个值有可能导致p-value超过0.05。

这并不意味着无法使用和不应使用统计技术。恰恰相反，它们仍然是分析有噪声数据的最好工具的代表。只是因为现在的主流还是离散实验，所以分析时应该多加小心。

1.2.2　松散结构

与其他许多数据集相比，流数据经常显得结构松散。这有多方面的原因。尽管松散结构并不是流数据独有的，但在流环境下似乎比在其他情况下更加普遍。

部分原因似乎在于流环境下所感兴趣的数据类型。流数据的来源多种多样。有些数据来源有严格的结构，然而还有许多数据来源包含了任意的数据载荷。特别是，社交媒体数据流会包含从世界各地发生的事件到周日晚上布鲁克林最好的比萨这样五花八门的数据。更难以应付的是，这些数据是自然语言形式的。

另一个原因是流数据项目的“厨房水槽”心态。多数项目非常年轻，探索的是未知的领域，因此将尽可能多的不同维度加入数据中是有意义的。随着时间的推移，这会有所改变，也开始使用容易修改的格式，例如JavaScript对象表示法（JavaScript Object Notation，JSON）。普遍的模式就是对实际感兴趣的事件尽可能多地采集数据。

最后，数据采集的实时属性还意味着每个维度都可能在任何给定时刻存在或消失。例如，将IP地址转换为地理位置的服务可能临时中断。对于批处理系统，这不会带来什么问题，服务再次上线时就可以重新进行分析。流处理系统则不然，它必须能够处理每一维上的变化，并尽力做到最好。

1.2.3　高基数的存储

基数指的是一部分数据可能的不同取值个数。形式上，基数指的是集合的大小，它可以用于数据集的各个维度以及整个数据集本身。高基数数据经常具有“长尾”特征：对于一个给定的维度（或维度组合），存在一个包含不同数据状态的小集合，这个小集合十分常见，通常占据所观察数据的绝大部分，其他数据状态只占很小的比例，从而形成一个“长尾”。

高基数特性在流系统和批处理系统中都很常见，但在流环境下对高基数的处理要困难得多。批处理环境下通常可以对数据集进行多遍处理。第一遍常用来确定数据的高基数维度，并计算构成大部分数据的状态。这些常见状态可以单独处理。剩下的状态则合并到一个名为“其他”的状态中，这个状态通常会被忽略掉。

流环境下，数据通常是单遍处理的。如果事先知晓常见的案例，就可以将这些案例放到处理步骤中。长尾部分也可以合并到“其他”状态中，分析就可以像批处理那样进行。如果已经对先前的数据集进行过批处理的考察，那就可以用来帮助流分析。然而，当前数据的常见状态是否也是未来数据的常见状态？这往往是未知的。实际上，混合状态的变化可能恰恰就是我们感兴趣的指标。更常见的情况是，没有以前的数据可以用来进行分析。这种情况下，流系统必须试图以数据本来的基数进行处理。

这对于处理和存储都是巨大挑战。对大规模集合做任何事情，必然要花时间处理涉及大量不同状态的种种问题，还需要线性大小的空间来存储各个不同状态的信息。流处理环境下对存储空间的要求要比批处理环境下更加严格，因为与批处理环境不同，流处理通常必须使用快速的主存储器，而不是速度缓慢的硬盘这样的第三级存储器。随着高性能固态硬盘

（SSD）的出现，这种情况有所缓解，但SSD仍然要比主存储器慢好几个数量级。

因此，对高基数数据的处理是流数据研究的一个主要课题。本书讨论了其中一些方法。这是一个活跃的研究领域，每天都会开发和改进很多解决方案。

1.3　基础架构和算法

本书旨在使读者具备从头到尾实现一个流数据项目的能力。没有基础架构的算法可能成为一篇有意思的研究论文，但无法造就一个成熟的系统。没有应用的基础架构多半只是浪费资源。

一心只关注算法或基础架构，指望着“建好了就自然有人来用”，这是行不通的。实际的系统必须同时实现用来支持系统的算法和基础架构。有了一个例子，其他人就能够看到各模块是怎么组合在一起的，然后将他们自己感兴趣的领域加入基础架构中。构建基础架构时要记住重要的一点（这值得反复重申），那就是我们的目的应当是使基础架构和算法可以被企业内（或者全世界）的各种用户所访问。只有当我们的项目能让人们使用、改进和扩展，这样的项目才算得上成功。

1.4　总结

总之，互联网规模的数据采集的兴起将实时处理框架与“传感器平台”连接起来。最初，这些传感器平台是完全虚拟的，例如系统监控代理，或是网站和以广告为目的的处理框架之间的连接。随着互联网的广泛普及，这转移到了实体世界，从而使得覆盖广大工业界的海量数据采集得以实现。

一旦数据可以实时获得，数据的处理必然也要是实时的。否则，既然从单次的角度看，批量采集很可能仍然要更便宜一些，实时采集又有什么价值呢？这就带来了一系列严峻挑战。使用本书中的基础架构解决方案与计算方法的组合，可以在很大程度上克服这些挑战，从而能够实时处理当今乃至未来系统采集的日益增多的数据。

第一部分　流分析架构

这一部分内容包括：

第2章　实时流架构设计

第3章　服务配置和协调

第4章　流分析中的数据流程管理

第5章　流数据的处理

第6章　流数据的存储

第2章　实时流架构设计

实时架构本质上是分层系统，该系统依赖于若干松散耦合的系统来达到目标。使用这种结构的原因有很多，例如保持系统在不可靠环境下的高可用性，对服务的需求，以及对架构本身成本结构的管理等。

本书第一部分介绍这类架构的各组件涉及的软件、框架和方法。作为该架构的蓝图和基础，本章首先介绍架构的各个组件，这些组件通常（但不总是）对应了物理或虚拟的独立机器实例；随后，我们以实时架构的主要特性（高可用性、低延迟、水平可扩展性）为背景对它们进行讨论。

本章还花了一些时间讨论建立实时架构的程序语言。我们假设读者已经在一定程度上熟悉本书示例所使用的Java和JavaScript语言。书中的许多软件包都是用Java虚拟机实现的，不过并非一定要用Java。作为Web的通用语言，JavaScript会在后面几节频繁使用，用于实现数据接口。

最后，本章为规划流架构试点项目的读者提供了一些建议。书中许多软件包可以在任何环境中发挥作用，在这个意义上说，可以将其视为是可互换的，但是，它们也都是优点与缺点并存。基于我们的实践经验，这自然就意味着在一些环境下，某些软件包要比其他软件包更好用。本章最后一节给出了以前成功的解决方案带来的一些启示，你可以将这些启示作为未来工作的起点。

2.1 实时架构的组件

本节介绍现代实时架构中常见组件。并不是所有系统都会有所有这些组件，在系统建设初期更是如此。合理目标应该是，建立适合初始应用的系统，使其具备可扩展能力，将来再加入缺少的组件。

第一个应用完成后，流基础架构的其他应用必然会致力于满足各自的需求。实际上，某些组件很可能会有多个实现。这种情况最常出现在存储组件中，因为各种存储组件产品的性能特征范围很宽广。

本节涵盖了各个组件，以及关于每个组件的开发和未来发展的一些问题。各个组件的详细内容可以在第一部分末尾的相关章节中找到。

2.1.1 数据采集

本书讨论的实时应用的最常见环境都基于TCP/IP网络。因此，数据采集通过建立在TCP/IP网络上的连接来进行，通常使用HTTP这样的通用协议。这种形式的数据采集最明显例子就是对延迟敏感的大型网站，它们通过接地域分布服务器（即边缘服务器）来缩短终端用户的延迟。

由于网站是大规模数据采集的最早用例，因此Web服务器所使用的日志格式成为日志文件格式的主流也就不足为奇。遗憾的是，多数Web服务器遵循美国国家超级计算应用中心（National Center for Supercomputing Applications，NCSA）公共日志格式（Common Log Format）和后来的万维网联盟（World Wide Web Consortium，W3C）扩展日志文件

（Extended Log File Format）格式，这两种格式从未打算支持丰富的数据载荷（因此在这方面表现很差）。

新式系统支持的日志格式更加丰富，其中JavaScript对象表示法（JavaScript Object Notation，JSON）是其中最流行的格式。JSON之所以流行，是因为它容易通过扩展来表示丰富的数据结构。JSON还比较易于解析，同时它的库也是广泛可用的。JSON一般用于在客户端Web应用与服务器端之间传送数据。

JSON的灵活性也带来了弊端：处理时的验证问题。缺少模式（schema）定义，以及不同的包容易产生结构不同但语义相同的结构，这都意味着处理JSON的应用往往包含大量的验证代码，以确保输入的合理性。JSON实质上是将消息和模式一起发送，这导致数据流需要较高的带宽。数据流压缩能够改善这种情况，经常能够达到80%以上的压缩率，但这也说明还有进一步提高的空间。

如果数据结构良好，对问题理解得又很到位，那就可以尝试使用结构化有线格式（structured wire format）。其中最流行的两个是Thrift和Protocol Buffers（通常简称Protobuf）。前者由Facebook开发，后者则由Google开发，两者的设计非常相似（它们还共享某些开发人员，知道这一点你就不会感到奇怪了）。它们使用一种接口定义语言（Interface Definition Language，IDL）来描述数据结构，IDL会被翻译成各种输出语言能用的代码，该代码被用来对有线传输的消息进行编码和解码。这两种格式都提供了一种机制来扩展原始消息，从而能够添加新的信息。

另一个相对小众的选择是Apache Avro格式。从概念上讲，它与

Protocol Buffer或Thrift非常相似。它也是用IDL来定义模式，其模式正好也是JSON格式，这与Thrift或Protocol Buffer很像。Avro倾向于使用动态编码器和解码器，而不是代码生成的方式，但其二进制格式却与Thrift非常类似。两者的重要区别在于，除了二进制格式之外，Avro还可以对其模式的JSON格式表示进行读写。JASON的非正式模式往往会被显式地声明为Avro模式，这样，我们就可以在已有的JASON表示和更简洁、定义更良好的二进制表示之间自由转换。

对大部分应用来说，数据采集过程是直接集成在边缘服务器中的。对于新服务器来说，所集成的数据采集机制很可能可以与下一节会讲到的数据流程机制直接通信。旧服务器是否能直接集成数据流程机制则很难说，可以根据具体情况进行选择。这些服务器通常是针对特定应用的，因此本书对此没有花费太多篇幅，只是讨论了数据流程组件的直接写入机制。

2.1.2　数据流程

企业内部的数据采集、分析和报告系统通常以不同的速率不断膨胀。例如，如果输入流量保持稳定，但分析的深度有所加强，那么尽管所采集的数据没有变化，分析基础架构还是需要更多的资源。为了应对这种状况，就要将基础架构分成采集、处理等多个层次。许多时候，这些层之间以专有（ad hoc）方式通信，这种环境中的每个应用都使用自己的通信方法来与其他层集成。

实时架构的一个目标就是将这种环境统一起来，至少在一定程度上使得应用的构建和分析更加模块化。数据流程系统（本书中也称为数据移动

系统）就是这其中的一个关键部分。

这些系统用单个统一的软件系统替代专有的、针对特定应用的通信框架。用来替换的软件系统通常是分布式的，从而易于扩展，并可以处理像多数据中心部署这样的复杂情况，同时，对数据的生产者和消费者也开放了通用接口。

本书讨论的系统主要是业界所称的第3代系统。“第0代”系统是紧耦合的专用通信系统，用来将应用分解为特定应用层。

第1代系统解除了这种耦合，通常使用某种日志文件系统来采集特定应用的数据，将其放入文件中。随后，这些文件一般会被收集到一个中央处理地点。接着，定制的处理器会使用这些文件来实现其他层。目前为止，这是最流行的系统，因为通过实现“至少一次”的交付语义保证了可靠性，同时对批处理应用又足够快。最初的Hadoop环境实质上就是针对这种用例进行优化的。

基于日志的系统主要的缺点在于速度太慢。日志文件一旦“展开”就要全部处理，因此数据必须以批量的形式采集。第2代数据流程系统认识到可靠传输并不总是优先目标，因此开始为系统间的数据移动实现远程过程调用（RPC）系统。Scribe和Avro这样的第2代系统，尽管可能为改善可靠性预留了一定的空间，但总的来说都倾向于牺牲可靠性以换取更快的速度。对许多应用来说，这种取舍是完全可以接受的，类似的模型已经在像Syslog这样的系统监控软件中应用了几十年。

第3代系统兼顾了第1代日志模型的可靠性和第2代PRC模型的高速

度。这类系统中，用于数据的生产者和消费者的数据层都有一个实时接口，数据以分散的消息进行交付，而不是像第1代日志系统那样批量进行。实际情况下，为了提高性能，这通常以包含上千条消息的“迷你批量”形式交付。

不过，为了保证与基于日志的交付系统相同的“至少一次”交付，这类环境会实现一个中间存储层。为了维持必要的性能，这种存储层会水平扩展到多个不同的物理系统，由系统的数据生产者和消费者方的客户端软件负责协调（coordination）工作。

这类系统中最早的就是用来处理大量数据负载的队列系统，ActiveMQ就是其中之一。队列系统通过队列机制形成了消息“总线”，将架构的不同组件解耦，使得开发人员不需要再关心通信任务。队列系统的缺点是需要保持队列语义，即数据消费者收到消息的顺序要与消息提交的顺序一致。一般来说，在分布式环境下这种语义并不是必要的，如果需要，由客户端软件来处理通常比较好。

鉴于多数情况下并不需要队列语义，第3代数据流程系统的最新成员Kafka和Flume应运而生。它们在很大程度上摒弃了排序语义，同时仍然保留了分布式系统和可靠性保证的概念。这使得它们几乎可以提高几乎所有应用的性能。Kafka还有一点值得注意，它是明确针对大规模多数据中心中的数据流程处理而设计的。

2.1.3　数据处理

Map-Reduced的（Hadoop是其最流行的开源版本）大行其道，并不是因为它是改变游戏规则的技术。实际上，1890年人口普查使用的霍列瑞斯穿孔制表机（Hollerith tabulation machines）就使用了与Hadoop及其他系统所实现的Map-Reduce过程相似的机制。

注意　霍列瑞斯后来创立了一个公司，这个公司就是IBM的前身之一。

引入分布式计算思想以及在数据本地进行计算以提高性能的重要性，也与Map-Reduce无关。高性能计算及大规模计算领域早就知道，将软件尽可能靠近数据往往效率更高，该领域的分布式计算软件也是应有尽有。

真正由Map-Reduce做的事情是，只要应用开发人员能够用这种独特的模式表述问题，计算的高效组织就由Map-Reduce负责，开发人员不需要参与。我们早就有表格形式的大量数据，但是一直以来，都是由特定的应用来决定怎样将数据划分到处理单元，以及怎样在其中移动数据。第1代日志采集系统实现了以通用方式采集大量数据。而Map-Reduce，特别是Hadoop，使得处理的分布化更加容易，从而不再需要针对特定应用的代码，或者昂贵的软硬件解决方案。

随着第2代数据流程框架的引入，类似的事情也发生在实时领域。以往封存在日志文件中的数据，现在能以低延迟的方式获得。与最初的批处理系统类似，第一批系统往往是针对特定任务实现的。例如，Twitter的Rainbird项目就是用Cassandra数据库实现的层次式流统计系统。

目前的第2代系统已经超越了针对特定任务的实现，开始为流计算的

组织提供通用服务。这些系统，例如Storm（也是Twitter的项目，但最初是由BackType公司开发的），通常提供一个有向无环图（Directed Acyclic Graph，DAG）机制，用于对计算进行组织，以及在计算的不同部分之间传送消息。

这些框架还处于比较初级的软件项目阶段，仍然经常要进行大量开发工作。它们也缺少容易理解的计算模式的指导，这就意味着只能随着项目的进展进行探索。随着开发工作的不断深入，批处理系统与流系统之间的区别很可能会变得模糊。Map-Reduce模型已经逐渐具备了实现更复杂计算的能力，从而能够支持项目中的交互查询要求，如Cloudera公司的Impala项目和Hortonworks公司负责的Stinger项目。

2.1.4　数据存储

实时处理和分析可以选择的数据库十分丰富，甚至达到了令人应接不暇的地步。尽管传统的关系数据库能够并且已经在流系统中使用，但通常我们还是选择所谓的“NoSQL”数据库。这种选择有许多原因，最主要的原因是性能优先，而不是传统关系数据库所要求的ACID特性（原子性、一致性、隔离性、持久性）优先。尽管ACID特性往往也会在某种程度上得到满足，但很少得到“NoSQL”数据库的完全支持。

与关系数据库相比，“NoSQL”数据库通常还缺少丰富的模式和查询语言。NoSQL得名于这样的事实：这类数据库的查询语言通常要比SQL（标准查询语言，Standard Query Language）标准严格得多。一旦有任何形式化的模式机制存在，这种简化的、更严格的查询语言通常伴随着更严格

的模式。

NoSQL数据库最常见的形式是各种键-值对（key-value）的持久存储，其范围从单机的主-从数据存储（例如Redis），到完全分布式的、满足最终一致性的存储（例如Cassandra）。其基本的数据结构是键，键值为任意长度的字节数组，但多数在核心实体上还建立了某种层次的抽象。其中某些（例如Cassandra）甚至对抽象进行扩展来提供类SQL语言，以包含模式和类似的语句，不过SQL语言的许多特性它们还是不支持。

NoSQL数据库还包括各种混合数据存储，例如MongoDB。与键-值对存储不同，MongoDB是一种索引文档存储。从许多方面来说，它更接近于Google这样的搜索引擎，而不是关系数据库。与键-值对存储相似，它的查询语言很有限。它没有模式，对具有丰富结构的文档则使用优化的JSON表示法。与多数键-值对存储不同，它还对文档之间的引用进行抽象。

伴随着查询语言和模式维护的简化，NoSQL放松了对前文提到的ACID特性的要求。在这些数据存储中，原子性和一致性尤其是经常被牺牲的对象。通过将一致性约束松弛为“最终一致性”，这些存储减少了簿记（bookkeeping）指令，并用更为简单的模型将存储分布到物理上相距很远的大量机器中，来换取性能的提高。在实际情况中，对于流应用，一般来说，不必保证每个客户端同时具有相同的数据视图。主要问题发生在两个物理上分离的数据库副本试着修改相同的状态时。这是个棘手的问题，但还是可以解决的，第3章会对此做详细介绍。

放松原子性要求通常也会带来数据存储的性能提升，而性能最大化就

是所有这些数据存储的最终目标。它们多数以某种轻量级的方式，通常是计数器（counter）类型的特例，来保持原子性。维持原子操作的计数器并不难，其用例也很常见，因此多数NoSQL数据存储都实现了某种形式的计数器。

本书讨论的各种存储都有优缺点。传统关系数据库甚至也在实时应用中占有一席之地。在哪个数据存储上应用运行得更好，取决于要实现的目标，用一种数据存储来满足所有目的是不现实的。本书的目的是讨论这样一种基础架构，它用实时存储作为记录系统（system of record）的良好近似，而记录系统可能是基于批处理系统和关系数据库的。然后，基于它们在各自问题上的性能特性，对数据存储进行选择。我们希望该基础架构能够确保这些存储都能持有实质上相同的数据，并针对各自的用例进行了优化。

接受了数据需要存储于不同的数据存储中这个现实之后，与批处理系统的集成也就是自然而然的事情了。流架构非常适合可以接受结果有一定误差的任务。取决于所需要的计算，这样的误差甚至可能是蓄意而为，这会在第9章和第10章中讨论。但是，往往最终必须对数据的计算进行解析和审计，这样的任务更适合批量计算。理想情况是，流架构负责快速获取数据，一旦批处理系统可以产生数据，就改由批处理系统负责。Storm的主要创始人Nathan Marz将这个命名为Lambda架构。在Lambda架构的系统中，中间件的用户界面或其他部分负责从相应的存储中获取需要的数据。随着批处理和实时系统的持续融合，通过Apache YARN项目这样的基础架构方面的改善，这种理想情况应该会越来越常见。

2.1.5　数据交付

多数实时应用的交付机制是基于Web界面的。这有很多好处。它们易于实现，并且由于是轻量级的，因而便于原地部署。在交付过程中，有许多框架可资利用。

最初这些仪表板就是带有META刷新（refresh）标记的网站，每几分钟重新加载交付过程的静态表示。许多系统监控工具，如Graphite，仍然使用这种方式。在应用数据的变化通常很迟缓时，这种方式的效果足够好。在需要支持多种环境时效果也不错，因为这项技术差不多与Web一样古老。

2.1.5.1　Web通信方法

对于将数据交付给Web浏览器来显示，现代应用有许多其他选择。更新Web页面的一个略快版本是使用多数Web浏览器带有的XMLHttpRequest（XHR）属性，将数据而不是数据镜像加载到浏览器。这类技术也称为AJAX，即“异步JavaScript与XML”。AJAX分离了数据层和表示层，相比旧模型有所进步。它使得应用可以根据环境对数据的渲染进行裁剪，并且往往能减少需要传输的数据量。

通过XMLHttpRequest传输数据，与页面刷新类似，本质是建立在轮询之上的。数据更新不会快过轮询的频率，这限制了新信息交付给前端的速度。如果事件相对较少，例如通知系统，大量调用浪费在确保“没有新闻就是好新闻”之上。随着对更加实时的通信需求的增长，通过使用长轮询这样的技术，有许多“高技巧程序”被开发出来，以使浏览器和服务器之

间的交互看起来更加迅疾。这些方法统称Comet，它们将真正的流式交付引入了浏览器，实现了真正的实时接口。

目前使用的有两个标准：服务器推送事件（Server Sent Event，SSE）和Web Socket。SSE为从服务器到客户端的单向流数据实现了兼容HTTP的协议。对于仪表板这样的主要消费数据的应用，这是很理想的。其他方向的通信则通过常规的AJAX调用来处理。与之相对，Web Socket实现了客户端和服务器间双向交互的更复杂协议。这种协议并不兼容HTTP，但被三个主要的桌面浏览器（IE浏览器、Chrome和Firefox）以及多数移动浏览器所支持。Internet Explorer不支持SSE，但SSE的HTTP兼容性意味着，对于不支持本地接口的浏览器，可以用轮询的老办法，不用编写额外的服务器端代码。

2.1.5.2　Web渲染方法

通过将数据发送到客户端，应用程序还可以利用内置在Web浏览器中的数据渲染组件：可伸缩矢量图（Scalable Vector Graphics，SVG）和Canvas绘图。其他绘图技术，例如只有IE浏览器支持的微软矢量标记语言（Vector Markup Language，VML）以及Google Chrome浏览器支持的WebGL标准，多年来也广泛使用。这些技术都试图将3D渲染标准引入Web浏览器。

SVG格式早在20世纪90年代就已经出现，但直到当前这一代的Web浏览器出现才得到了很好的支持。通过使用与PostScript和可移植显示格式（Portable Display Format，PDF）相似的绘图模型，SVG使得Web页面具备了与分辨率无关的图形渲染能力。与HTML类似，SVG通过使用能够

显示矩形、圆和任意轨迹的标记语言来指定图形元素。对这些图形元素，可以用层叠样式表（Cascading Style Sheet，CSS）来指定样式，通过对文档对象模型（Document Object Model，DOM）的操作来制作动画效果。原始的SVG最初只是通过繁杂的插件接口来支持的，包含在Firefox 2.0以及Safari和Chrome浏览器所使用的WebKit数据渲染引擎的早期版本中。现在，所有主流浏览器的最新版本和多数移动浏览器的实现都支持SVG。这样广泛的支持以及与分辨率无关的特性，使得SVG成为跨各种平台的图形渲染的理想选择。

Canvas元素最初出现时是苹果公司Safari浏览器的一部分，随后被包含在作为HTML5标准基础的WHAT-WG提案中，它也赢得了浏览器的广泛支持。Canvas元素为图像渲染提供了绘图层，而不是文档对象模型。绘图模型本身非常类似于SVG，因为它也借鉴自PostScript和PDF。不过，它只维护实际的绘制像素，而不是用来渲染像素的对象。这使得canvas元素的作用更像传统的HTML图像元素，同时它在某些情况下的性能堪比SVG。在图形学领域，Canvas通常作为“即刻使用模式”的实现方案，而SVG是“保留使用模式”的实现方案。

2.2 实时架构的特性

前一节讨论的实时系统的所有组件都有三个关键特性，使得它们可以在实时流环境运行，这三个特性是：高可用性、低延迟和水平可扩展性。没有这些特性，实时系统就会存在根本上的缺陷，无法正常扩展。本节在实时应用的背景下讨论这些特性的含义。

2.2.1 高可用性

可以说，整个系统对于高可用性的需求是实时流应用与更常见的批处理或商业智能应用之间的关键区别。如果后者出现几分钟甚至几小时的故障，不太可能影响运营。（甚至往往不会引起用户的注意。）实时系统则不然，它们对这类运行中断很敏感，甚至对有计划的维护窗口也可能很敏感。

高可用性未必适用于栈的所有组件，比如对交付机制就不适用，但对于采集、流程和处理系统通常都非常重要。为了确保高可用性，栈中的这些系统多数诉诸两种策略：分布化和复制。分布化（distribution）意味着使用多个物理服务器将负载分散到多个端点。例如，如果数据采集系统中的其中一台机器出现故障，另一台就可以捡起接力棒，直到原来的机器恢复正常或者被替换。

当然，高可用性并不单指服务的可用性，还包括所处理数据的可用性。数据采集系统通常并不维护多少本地状态，因此不需要做什么恢复工

作就可以被即刻替换。数据流程系统的故障则不然，因为它的某些数据会无法使用。多数系统会采用某种数据复制机制来克服这个问题。

数据复制（replication）的基本思想是，系统不是将单份数据都写到单独一台机器中，而是写到多台机器中，寄希望于至少其中一台能够在故障中幸存。以关系数据库为例，它们经常会实现数据库复制，使得对主节点机器的修改复制到多个从节点机器上，同时有各种机制来保证，在数据可以由客户端从数据库读取之前，有若干个从节点机器必须有该数据。如果主节点机器由于某种原因宕机，某些数据库就会自动进行故障切换，令其中一个从节点机器成为主节点机器。这种故障切换通常是永久的，新主节点机器的身份是永久的，这是因为即使以前的主节点机器后来恢复工作，它也会丢失过渡期的数据修改。

在本书中提出的软件栈中，有的软件也使用同样的方法。Kafka数据移动系统使用主-从模式的复制，来保证即使发生故障，写到队列中的数据也仍然可用。MongoDB数据存储也使用类似的系统作为数据复制的解决方案。这两种情况下，都会进行自动故障切换，不过客户端可能需要先断线再重连，以识别新的主节点机器。客户端还要负责处理过渡期间还没有被主节点机器确认的数据修改。

还有一种NoSQL数据存储常用的高可用性方法，在这种方法中没有主节点。与主-从配置类似的是，所有修改都会写到分布式机器池的多个服务器中。通常会根据数据的特征，例如要写入的数据的主键值，选一台机器作为“首要”（primary）的写机器。随后该首要机器采取某种方式，将同样的值写到根据该键值确定的多台其他机器中。用这种方式，首要机器总

是首选试着读写数据，如果首要机器出现故障，就由其他机器按照顺序来尝试。本书中讨论的Cassandra数据存储就实现了这个基本过程。客户端软件实现这种多写入机制也很常见，这种技术经常用于分布式哈希表（Distributed Hash Table，DHT）来保证高可用性。与主-从架构不同的是，这种方法的故障服务器恢复会比较复杂，这是它的缺点。

2.2.2 低延迟

对大多数开发人员来说，“低延迟”指的是为给定连接提供服务的时间。低延迟通常是必要的，这是因为每天只有那么多时间（86400秒），一台机器处理单个请求需要的时间越少，它所能服务的请求就越多。

对于实时流应用而言，低延迟的含义略有不同。它指的不是单个请求的完成时间，而是事件在系统“边缘”某个地方发生的时间，与事件可以由处理和交付框架处理的时间，这两者的时间差。另外，它还隐含了一个没有明确表述的意思，即各种事件的延迟之间的差别是非常稳定的。

举例来说，在分钟级运行的批处理系统中，有的事件的延迟是非常小的，具体来说，就是那些在处理过程开始前就已经进入处理批次的事件。而在处理循环开始之后才进入处理批次的事件则会有很高的延迟，因为它们需要等待下一个处理批次。出于现实考虑，许多流系统也与所谓的有效批次（effectivity batch）一起工作，但这些批次非常小，以至于和所监控或处理的问题的时间量级相比，这种小批次的首、尾元素之间的时间差别微乎其微，通常是毫秒数量级。

系统的数据采集组件大多考虑低延迟的第一个定义。一台服务器所能处理的连接越多，需要用来处理负载的服务器就越少。为了满足前一节的高可用性需求，通常需要不止一台边缘服务器，但对于实时系统来说，能够减少边缘服务器的数量对于降低成本通常大有裨益。

数据流程和处理组件更关注低延迟的第二个定义。这里的问题就是，减小事件到达处理组件与可以提供给消费者之间的时间差。这里的关键是要尽可能避免同步，尽管有时为了维持高可用性，这难以避免。例如，Kafka在第一个公开发行的版本0.7和本书所讨论的版本0.8之间，加入了复制功能。它还在Producer API中加入了一个响应，用来通知客户端消息确实到达了主机。在此之前，这都是默认的。不出所料，处于这两个版本之间的Kafka的延迟果然有某种程度的增加，因为此时，数据在可以提供给数据消费者之前，必须经历“同步复制”。在多数实际情况下，多几秒钟的延迟可能并不要紧，但在实时流应用中，我们经常要在速度和安全性之间做权衡。如果可以安全地丢弃数据，就可以得到很小的延迟。反之，除非物理学有了相关进展，降低对低延迟的要求就不可避免。

2.2.3　水平可扩展性

水平可扩展（horizontal scaling）指的是，向处理特定问题的服务器集群加入更多物理独立的服务器来提高性能。与之相对的是垂直可扩展（vertical scaling），即向一台服务器加入更多资源来提高性能。多年以来，垂直可扩展一直是开发人员不必求助于外来硬件解决方案来解决性能问题的唯一可行方法，但是随着低成本的高性能网络的出现，以及软件技

术的进步，处理能力几乎线性增长，基本上投入双倍的硬件就可以获得双倍的性能。

成功进行水平扩展的关键在于，限制系统间所需的协调。这里的限制既适用于系统间不同实例间的通信，也适用于系统间相同实例的通信。解决这个难题的一个主要机制是分区技术，即将任务划分到不同机器中。这就保证了特殊类别的任务由众所周知的机器完成，不再需要向机器询问其担负的任务。

一旦需要协调，问题就会变得非常复杂。多年以来开发出了许多算法，用来使松耦合的机器可以协调和处理网络故障这样的事件。实践已经证明，这些算法实现起来都很不容易。幸好现在我们可以通过协调服务器将这些算法提供为“服务”。其中最流行的是由Yahoo！开发的ZooKeeper系统，我们将会在第3章对此做详细讨论。本书许多其他的框架和软件包都会用到这个系统。它极大地简化了在一组机器之间进行高效协调的任务。

2.3 实时编程语言

当今世界的编程语言五花八门，但只有少数几种广泛用于实时流应用的实现。部分原因是流应用的核心软件包是用专门语言实现的，故而该软件包的客户端库当然就主要考虑其本地语言。另外，这些语言的速度和易用性使得它们很适合于实现实时应用。下面我们对实时应用中经常碰到的几种语言加以介绍。

2.3.1 Java

Java语言是在20世纪90年代中期公之于众的，用Java语言编写的软件号称能够“一次编写，任意地点运行”。刚开始时，它用来以小应用程序（applet）的形式将Java软件嵌入到Web页面中。其目标是使得Web服务器作为“胖”客户端-服务器应用的分发机制。出于各种原因这个目标从来没有真正实现，现在许多Web浏览器却默认禁用Java。它最初追求的功能多数转由Adobe的Flash和Web页面本身来提供，后者是由于近年来所有Web浏览器都内置了JavaScript引擎，这个引擎的功能足以实现丰富的应用。

尽管Java在客户端落败，但由于它比主要的竞争对手C++更易于部署，因此在Web应用的服务端占据一席之地。由于很早就引入了Java数据库连接（Java Database Connectivity，JDBC）标准，在涉及数据库连接时，Java尤其容易部署。多数Web应用本质上都是对数据库的连接，因此Java成为服务器的常见实现语言。

Java拥有大量的软件库，其性能改进尽管缓慢但很稳定，这使得它成为许多早期的大数据应用，特别是Hadoop的自然而然的选择（不过，Google内部的Map-Reduce应用倒是用C＋＋编写的）。用Java来编写软件栈中的其他部分是有道理的，因为用Java编写的Hadoop形式占据了前端和后端。

由于本书中的多数软件包是用Java编写的，或者至少是以运行在Java虚拟机（Java Virtual Machine，JVM）之上为目的，因此我们主要以Java作为本书示例的编程语言。这与当初将Java作为软件包的实现语言具有相同的原因。Java语言的软件库多种多样，并且因为Java具有类似的许多构建管理系统，这些库都很容易得到。

本书使用了称为Maven的构建系统，来管理软件包和它们之间的依赖关系。Maven很好用，因为它具备软件包的分发能力，从而能够通过它自身的中心资源库，或者软件库作者自己维护的资源库获取软件包。这些资源库自行维护这些软件，同时还会描述软件包的依赖关系。然后Maven就能获取需要用来自动构建软件包的全部相应的软件。

2.3.2　Scala和Clojure

上一节我们提到过，多数软件是用Java构建的，或者是“在Java虚拟机上运行的”。这是因为某些软件尽管实际上并不是用Java语言编写，但编译后是在JVM上运行，并且这些语言能与Java软件包交互。

有许多非Java的JVM语言，但在实时应用开发中最流行的两个就是

Scala和Clojure。本书中，Scala用来实现用于数据移动的Kafka软件包，以及与Kafka一起用于处理数据的Samza框架。Scala长期以来是一种学术语言，目前仍然主要在大学里开发，但其丰富的标准库对高性能服务器应用的开发人员们有着很大吸引力。与Java类似，Scala是一种强类型的面向对象语言，但它也包含了标准Java语言所不具备的许多函数式编程语言的特性。有趣的是，Java8似乎吸收了Scala以及其他函数式语言的几个更有用的特性。

Scala没有可以与之类比的编程语言，Clojure是一种JVM语言，它基于历史悠久的Lisp语言。多年以来，有许多为Java编写的Lisp解释器，但Clojure是编译型语言，并且能在其程序中直接使用Java库。Clojure用来实现本书讨论的Storm流数据处理框架。Lisp风格的语言十分适合于实现特定领域的语言，这种语言都是用来实现特定任务的小型语言。看起来这对Storm的特定领域语言Trident的开发也有影响，这种语言通过Storm中的各种处理步骤来定义数据流程。

2.3.3　JavaScript

JavaScript当然是Web的通用语言。JavaScript内置在所有的Web浏览器中，由于浏览器开发人员之间的竞争，即使单独用作Web应用的实现语言，它的速度也已经足够快了。

JavaScript和Java之间的联系，除了包括名字上相近这个营销噱头，就只有表面上语法的相似性。二者都继承了C/C＋＋语言的大量语法，但JavaScript更接近于Clojure和Scala。

与Scala和Clojure类似，JavaScript也是函数式语言。在Java中，对象是包含数据和函数的实体，后者也称为方法（method）。这两种实体是独立的，Java编译器对其处理的方式有很大不同。Java编译方法时，会将其转换为称为字节代码的指令，随后除非被其他方法调用，该方法多半会从Java环境中消失。在函数式语言中，对函数的处理方式则与数据相同。它们可以像整数或字符串一样存储在对象中，可以从函数中返回，以及传递给其他函数。这种特性在许多不同的客户端JavaScript框架中大量使用，其中包括本书用到的D3框架。D3框架的全名是数据驱动文档（Data Driven Documents），它用函数来表示用于操作Web页面的公式。例如，可以向对象的方法传递一个函数来说明某个矩形应该出现在页面的什么位置。这使得框架不用花费太多开销就可以响应外来数据。

在交付服务器的实现中，JavaScript也用得越来越多，它主要用在node.js项目（http://nodejs.org ）中。该项目将Chrome Web浏览器的JavaScript引擎V8与一个事件循环库结合起来。这使得本质上是单线程的服务器能够实现处理大量并发连接的事件驱动网络服务。尽管其效率无法与现代Java框架相比，但对于向前端进行数据交付的服务器的实现基本上是足够的。在对不支持SVG和Canvas这样现代特性的浏览器的向后支持方面，相比不基于JavaScript的服务器，基于JavaScript的服务器具有优势，因为可以用与客户端相同的代码库在服务器端实现。特别是，node.js拥有创建DOM的软件包，并可以在服务器端而不是客户端运行渲染服务来对基于SVG的可视化进行渲染。

2.3.4　Go语言

尽管本书没有使用Go语言，但它是高性能应用和实时应用开发领域的新宠。Go语言由Google的一个非常小型的团队所开发，它将C语言的简单性与高并发应用开发框架结合在一起。它的团队的核心成员包括了最初的UNIX团队和C语言开发团队一些成员，因此Go与C语言类似也就不足为奇。

Go语言在很大程度上仍然处于开发阶段，其编译器还比不上许多C编译器（甚至现在肯定还比不上Java编译器）。即便如此，在真实的Web服务器开发测试集上Go语言表现很好，并且本书中讨论的多个软件包，特别是Kafka和Cassandra，都已经有了Go语言客户端。这使得Go语言虽然不适合用来开发实时应用的处理和交付组件，但却是开发边缘数据采集服务器可以考虑的选项。

2.4 实时架构概览

剩下的内容讨论用来构建实时流系统的框架和软件包。每个软件都用于某个真实的应用。它们之间不应被视为存在竞争关系。它们各有自己的目的，因此，泛泛地对它们进行比较往往会不得要领。为此，本节没有对软件包横加比较，而是对于在给定环境下哪个软件包、框架或语言最好用，给出了简明的观点。

2.4.1 数据采集

许多情况下，对于数据采集并没有其他选择。边缘服务器是业已存在的系统，我们的目标是对服务器进行改进，或者将其与实时框架集成起来。

如果运气很好，边缘服务器恰好是为从外部采集数据而定制的，基于Java的服务器很可能就是最好的选择。Java绝对算不上是一门“性感”的语言。实际上，很多情况下它用起来可能令人不爽。但是Java已经有近20年的历史。人们对于它的缺点已经了如指掌，并且长期以来一直在努力提高其框架的性能。在实际应用中，如果再付出同样多的努力对其调优，Java的性能应该会越来越好。

其他类似于Scala或Clojure这样的编译型JVM语言的性能与Java类似。特别是在不使用这些语言的高级特性，如Scala的采集库时，更是如此。然而，专用的边缘服务器的目的是尽可能简单，以及迅速且安全地将

数据交付给数据流程框架。这对于各种复杂的Java框架是很难做到的。这些更为复杂的框架的设计目标是建立网站，而不是高性能的数据采集系统。因此，你应该使用轻量级的框架，而不是实现只有少量交互的网站所用的重量级解决方案。

正如2.3节所提到的，Go语言执行这种任务时往往表现很好。尽管Go语言并不完全具备调优良好的Java Web服务器所具有的性能，但它在实际测试中的表现仍然比几乎所有其他语言都要好。此外，据认为Go语言通常内存占用较小，并且有迹象显示，当真正有大量连接连到服务器时，Go语言比Java应用表现得更好。实例研究也表明，即使并发连接的数量超过5万，Go语言的仍然能够“正常”处理。其缺点是，几乎没有人知道怎样用Go编程，因此很难找到相关资源。

2.4.2　数据流程

如果要集成已有系统，那就应该选择Flume。它内置有对其他环境的接口套件，是向遗留环境加入实时处理系统的理想选择。它的配置和维护非常简单，能使你以最小的冲突将项目顺利启动并运行。

如果要对系统加以改进，或者从头开始创建新系统，那就很难找到理由不用Kafka。不用它只可能有一个理由，那就是所使用的语言没有相应的Kafka客户端。这种情况下，Kafka社区肯定会对针对该语言的客户端表示欢迎，即使是部分实现的（很可能是生产者（Producer）部分）也可以。即使加上了新的安全特性，Kafka的性能仍然非常好（与上一节的Scala形成鲜明对比），并且基本上该做到的也全部做到了。

2.4.3　数据处理

尽管Samza显示出了巨大潜力，可惜的是，对于多数首次尝试实时处理系统的人来说，它还是不够成熟。这主要因为它依靠Apache YARN框架进行分布式处理。据称，尽管YARN能够支持许多不同的计算类型，但现实情况是几乎所有文档都只考虑Map-Reduce模式。这使得Samza对于缺乏经验的用户显得太难。

已经熟悉YARN集群维护的用户成功使用Samza的概率会大很多，这样的用户可以考虑采用Samza来实现实时应用。Samza的设计在很多方面都比Storm简洁，它有对Kafka的本地支持，这使得将它与基于Kafka的环境进行集成比较简单。

对于初次使用的用户来说，Storm是更合适的框架。Storm也可以放在YARN集群中，但那并不是它典型的部署方式。本书中讨论的非YARN部署对新用户来说更易理解，也比较易于管理。唯一的缺点就是它不包含对Kafka或Flume的本地支持。第5章解决了这种集成问题，但插件的开发循环与主流的Storm代码差异很大，有可能导致发布时出现不兼容的问题。

2.4.4　数据存储

对于相对小规模的项目，例如概念验证或只有很小流量的应用，Redis是数据存储的显而易见的最佳选择。相比简单的键-值存储，Redis提供了更丰富的抽象，因此能支持更复杂的存储。它易于配置、安装和维护，基本上不需要进行多少维护（它甚至是各种基于云的提供商的一站式

方案（turnkey）的一个选择）。Redis还拥有一系列多种多样的客户端。

Redis有两个缺点：第一，它没有真正解决写操作的水平扩展性问题；第二，它能访问的主服务器的随机访问存储器（RAM）是有限的。类似于本书中的许多工具，它提供了一种主-从模式的复制，能够使用Redis Sentinel在发生故障时切换到其中一个副本。但是，这仍然需要所有的写操作集中到一台服务器。有几个客户端的项目，例如Twitter的Twemproxy，想要尝试解决这个限制，但目前为止还没有本地的Redis解决方案。还有一个正在进行的集群项目，但其稳定版本的发布还没有时间表。

如果需要存储的是海量的非频繁访问的数据，Cassandra是很好的选择。Cassandra的早期版本受累于其“查询语言”，这种“查询语言”本质上只是Cassandra内部数据结构之上的一个RPC层。该“查询语言”再加上一个（对于当时来说）难用的数据模型，使得Cassandra很不好用。随着新版Cassandra以及Cassandra查询语言（Cassandra Query Language，CQL）版本3的发布，这些问题多数已经得到纠正。加上对集群中数据分布的其他方面的修改，Cassandra已经成为一个使用和维护起来友好得多的环境。它也非常快，特别是当服务器装备了固态硬盘（Solid State Drive，SSD）的时候。虽然由于必须从磁盘中提取数据，其速度还不能媲美Redis，但也不会慢太多。

只有在要存储的数据有丰富的结构时（如地理数据流），MongoDB才是真正的首选。MongoDB提供了工具和索引来支持地理信息系统（Geographical Information System，GIS）数据，从而比Redis或

Cassandra都要高效。其他选项基本上都是传统的关系存储加上地理索引，因此MongoDB提供了一个高性能的选择。正因如此，专注于地理数据的Foursquare公司才会成为MongoDB早期的著名用户。

2.4.5　数据交付

几乎所有应用的交付迟早都会以Web应用的形式出现，至少在第一次迭代时是这样。本地应用，特别是移动设备的本地应用，向用户交付的体验肯定更加丰富和令人满意。这里还涉及一系列技术，有一些多数企业未必能立即掌握。因此，如果能找到移动开发人员，那就应该想尽办法聘用他们。对流应用来说，得益于本地体验，良好的平板电脑或手机界面要比Web界面更具吸引力。

即便如此，本地的移动体验仍然需要能交付数据的服务器。最好的方式是实现支持SSE的应用程序编程接口（API），由Web应用和本地应用使用。尽管Web Socket在跨浏览器支持方面要比SSE好一些，但我们选择SSE而不是Web Socket，因为SSE是基于HTTP的，而Web Socket的接口不是。SSE就是HTTP连接，这样就可以使用“polyfills”在旧式浏览器上模拟SSE。要是用Web Socket来做就困难得多，甚至可能做不到。与Web Socket的更复杂的协议相比，使用同样的记号将SSE集成到本地体验中也更简单。由于Web Socket主要目的是在Web浏览器内使用，故而不必在本地平台中作为库来出现。而标准的HTTP库通常可以进行简单修改，通过改变keep alive和timeout行为来支持SSE。

2.5 总结

本章对实时应用的发展做了总体概括。我们介绍了可称为层（tiers）的实时系统的各种组件，以及每个层可以使用的一些框架的名称。此外，我们也简要介绍了实时应用的底层工具：编程语言。正如我们在本章开头所提到的，我们假设读者对这些编程语言有一定了解，足以看懂本书后面的内容。

本书后面的内容将深入讨论本章所描述的每个组件。第3章首先讨论我们在第2.2.3节中提到的协调服务器ZooKeeper。这是一个关键软件，书中多个其他软件包都会用到，因此占据独立一章实至名归。协调服务器用来管理从边缘服务器到数据处理环境的数据流程。第4章介绍用来管理数据流程的两个软件包Kafka和Flume，正如前面提到的，这两个软件包应该根据需要选择使用。随后，数据就应该由Storm或Samza来处理，我们在第5章介绍这两个软件包。处理结果需要被存储起来，这有很多选择。第6章列出了其中较为流行的选择，这样这些流数据就可以由第7章中讨论的交付机制来访问。

第8章至第11章更关注在上述基础之上的应用构建问题。这几章介绍了第2.4.3节使用的聚集计算方法，以及用来对数据进行处理以达到特定目标的统计技术。第11章专门了介绍若干“高级话题”，包括流数据的机器学习问题等。

第3章　服务配置和协调

高性能计算早期的发展主要聚焦于纵向扩展（scale-up）架构。这些系统都具有多个处理器和一个共享的内存总线。实际上，现代多核服务器硬件设备在架构上与20世纪80年代的超级计算机硬件十分相似（尽管现代硬件设备通常抛弃了Cray公司的长条沙发设计）。

随着系统间的互联以及网络硬件的改善，这些系统开始变得更加分布式，最终演变为当今多数计算环境所使用的工作站网络（Network of Workstations，NOW）。然而，这些互联通常不具备模拟纵向扩展架构的共享资源环境所需的带宽，以及纵向扩展环境所必需的互联可靠性。分布式系统还引入了纵向扩展环境通常没有的新的故障模式。

本章讨论用来解决分布式环境下这些问题（特别是分布式应用中共享状态的管理问题）的系统和技术。首先讨论配置管理和协调系统。这类系统中最流行的（本书大部分软件使用的）就是ZooKeeper。该系统是由Yahoo！开发的，用来帮助管理其Hadoop基础架构，我们会对其做深入讨论，因为本书中的其他分布式软件系统都会用到该系统。

3.1 配置和协调系统的研发动机

多数分布式系统需要共享某些系统元数据，以及关于分布式系统本身的某些状态。元数据通常是某种配置信息。例如，可能需要在分布式服务器中存储各种数据库碎片的位置。系统的状态通常是用来对应用进行协调的数据。例如，主-从系统需要跟踪当前作为主服务器的服务器。而且，这是在不太可靠的环境下进行的，因此服务器还必须以某种方式对当前配置的正确性或协调操作的有效性进行跟踪。

在分布式环境下管理这些需求是众所周知的难题，经常会导致服务器的错误行为。但是，如果这些问题没有得到解决，则可能引发分布式系统的单点故障。对“离线”处理系统来说，这只是小问题，因为单点故障可以通过手工恢复来解决。在实时系统中，问题要严重得多，因为恢复过程可能会引入无法接受的处理延迟，或者完全错过处理（这更常见）。

这就直接引出了配置和协调系统的研发动机：提供一个系统范围的服务，以便正确、可靠地实现分布式配置和协调原语（primitive）。

与为多线程开发提供的协调原语类似，这些原语用来实现高级算法的分布式版本。

3.2 维护分布式状态

在单个应用中编写共享状态的并发代码是很难的。即使操作系统和开发环境提供了对并发原语的支持，线程和进程之间的交互仍是常见的错误来源。当并发扩展到多台机器时，问题会更加复杂。除了死锁、竞争条件等普通的并发问题之外，还有一大堆新问题要解决。

3.2.1 不可靠的网络连接

即使在控制最为良好的数据中心，网络也不如单机可靠。不同时刻的延迟可能差异很大，带宽也会随时间变化，连接可能会中断。在广域网中，“挖沟机事件”（Backhoe Event）可以导致先前一致的网络发生连接中断。对于并发应用来说，这样的事件（可能发生在单个数据中心中）是最严重的问题。

在并发编程中，两组系统之间的连接丢失称为“脑裂问题”（Split Brain Problem）。一旦发生，分布式系统必须决定怎样处理某些状态突然无法访问的情况。在实践中有很多策略，最常见的是进入降级状态。例如，当检测到连接丢失时，许多系统会禁止修改分布式状态，直到连接恢复为止。

另一个策略是允许其中一个分区保留全部功能，而对其他分区的功能进行降级。这通常通过quorum算法实现，这个算法需要在分区中有一定数量的服务器。例如，常见的策略是，保留全部功能的分区需要包含奇数

个进程。如果奇数个服务器分为两组，一组总是包含奇数个服务器，而另一组包含偶数个服务器。前一组保持全部功能，后一组变成只读或者只具备其他降级功能。

3.2.2　时钟同步

分布式系统经常需要某种时钟同步机制，初看起来这个问题似乎挺简单。对于某些应用，时钟同步可能需要非常精确。可惜的是，服务器中的硬件时钟并不完美，容易随着时间发生漂移。如果漂移过大，服务器可能会经历未来才发生的事件，从而引发有趣的事件。例如，对两类事件中时间戳的差值感兴趣的分析系统，可能会发现负的时间间隔。

多数情况下，服务器使用网络定时协议（Network Time Protocol，NTP）进行时钟同步。使用这种方式，有时机器间仍然会出现较大的时钟漂移，但通常都“足够接近”。不过，当机器无法与同一个NTP服务器集合同步时，也会出现问题。当一个环境拥有一个安全的内部域，这个域只有通过一个严格控制的网关才能与面向外部的域通信时，这种问题就时有发生。这时，内部的NTP服务器与外部用户所用的NTP服务器相比，可能有很大的漂移。某些情况下，例如对于要使用时间来管理授权（authorization）的应用程序编程接口（API），这种漂移可能导致服务发生严重问题。

可惜的是，除了“不要这样做”之外，对这个特殊问题，没有什么容易的解决方案。

3.2.3 不可靠环境下的一致性

除了网络连接不可靠和时钟不同步之外，进程本身也可能包含错误，导致它们有时生成不正确结果。对于这个问题，一些学术研究表明，这些进程的行为甚至可以是主动的恶意行为，尽管许多实际系统不会考虑这种情况。

不管出现什么问题，各机器维护的分布式状态必须是一致的。对此最著名的（不过不是第一个）算法是20世纪80年代末期提出的Paxos算法。

在Paxos算法中，想要改变共享状态的进程首先要向集群提交一个提案（proposal）。这个提案由一个单调递增的整数来标志，这个整数值必须大于该提交者（proposer）先前使用的所有标识符。

接下来该提案被传送给仲裁数量的其他进程，这些进程称为接收者（Acceptor）。如果给定的接收者收到了一个消息，该消息的标识符比它以前从提交者所看到的最大的消息标识符要小，那么它就丢弃这个陈旧提案（另一种做法是向提交者明确拒绝这个请求）。

如果接收者看到的消息是它从提交者那里看到的标识符最大的一条消息，它就更新从提交者那里收到的最大标识符值。同时返回先前的标识符，以及从提交者那里收到的相关值。这称为承诺阶段（promise phase）。因为接收者现在许诺，不再拒绝标识符小于该新标识符的所有请求。

当提交者从承诺阶段接收到足够的正面反馈后，它可以考虑请求将状

态改为已接受。提交者然后向接收者发送实际的修改，接收者随后将该信息传播给集群中的其他节点，这些其他节点称为学习者（learner）。这个过程也确定了集群的领导者（leader），即提案被接受的提交者。在实际情况中，这常用来实现服务器集群（例如，数据库应用集群）的领导者选举（leader election）。

基本的Paxos算法实质上考虑的是整体状态的更新。它也可以用来更新分布式状态表示的个别元素，但这容易引入大量开销，为了解决这个问题，提出了Multi-Paxos算法。

Multi-Paxos是经常在现实情况中实际实现的算法。它的提出基于以下事实：进程是相对稳定的，集群的领导者很少发生变化。这就允许算法在宣布某个进程为领导者之后，省去通信协议的准备-承诺（Prepare-Promise）阶段。当选出新的领导者时，必须再次重复准备-承诺循环，这样该算法在最差情况下就是基本的Paxos算法。

这些算法看起来似乎简单明了，但它们都是公认的很难正确实现的算法。因此，建议你直接使用本章给出的系统，不要尝试自己实现。本章介绍了两个此类系统：ZooKeeper和etcd。ZooKeeper是一个分布式配置和协调工具，用来协调本书中许多分布式系统。etcd是一个新系统，它提供许多与ZooKeeper相同的特性，具有基于HTTP的接口，使用称为Raft的新的一致性算法，这个算法用于降低Paxos的复杂性。

3.3 Apache ZooKeeper

ZooKeeper项目是由Yahoo！开发的，其任务就是提供上一节提到的原语。最初开发时，ZooKeeper是为了解决Yahoo！内部Hadoop环境中所实现服务的配置和协调问题，后来被开源并贡献给Apache基金会。

后来的发展证明，ZooKeeper是分布式系统开发的优秀基础架构组件。实际上，书中后面使用的许多应用都是用它来协调的。Kafka数据移动系统用它来管理服务器和客户端的元数据。它还是支持Storm数据处理框架的服务器管理特性的关键组件，在Samza数据处理框架中也具有类似的作用。

ZooKeeper并不强求一组特定的协调或配置特性，而是提供了一个低层的API，从而用基于分层文件系统的系统，来实现协调和配置服务。这些特性在ZooKeeper中常称为recipe。ZooKeeper将它们留给应用的开发人员自行实现。这样做的优点在于，能够简化ZooKeeper的代码库，但也会逼着应用开发人员来处理某些可能很难的实现问题。

还好有客户端软件库，例如后面讨论的Curator软件库，就精心实现了更复杂的recipes。本节首先介绍ZooKeeper本身的基本特性，然后介绍Curator软件库。

3.3.1 znode

znode是ZooKeeper的数据结构的核心。这些节点可以分层组织为树形

结构，并存放数据。由于ZooKeeper并不是海量存储工具，因此一个znode存放的数据量限制在约1 MB之内。这也意味着ZooKeeper不支持部分读、写或追加。当读写数据时，一次调用会传输整个字节数组。

znode API支持6种基本操作：

·create操作：以路径和可选的数据元素作为参数。如果不存在该znode，这个操作在指定路径创建一个存有数据的新znode。

·delete操作：从层次结构中删除znode。

·exist操作：以路径作为参数，应用程序不必读取数据，就可以用该操作来检查某个znode是否存在。

·setData操作：该操作的参数与create操作相同，对已有znode中的数据进行重写，但如果该znode不存在，则不会创建新的znode。

·getData操作：获取znode的数据块。

·getChildren操作：获取特定路径中znode的子节点列表。这个操作是为ZooKeeper开发的众多协调recipe的关键部分。

所有这些操作都遵从ZooKeeper的访问控制策略。这些策略很像文件系统的权限，规定了哪个客户端可以写入层次结构的哪个部分。这主要是考虑到多租户ZooKeeper集群的实现需求。

3.3.1.1　临时节点

znode可以是持久的，也可以是临时的。对于持久znode，只有使用

delete操作显式地从ZooKeeper中将它删除，才能将其销毁。

对临时节点来说，当创建这个节点的客户端会话与ZooKeeper集群失去联系或者主动终止会话时，这个节点也会被销毁。在前一种情况下，引发临时节点销毁所需要的失去联系的时间由心跳超时控制。

由于临时节点随时都可能被销毁，因此它们不能有子节点。这在ZooKeeper将来的版本中可能会有改变，但是在3.4系列版本中还是这样，临时节点可以是文件，但不可以是目录。

3.3.1.2　顺序节点

也可以将znode声明为顺序的。顺序znode被创建时，它会被赋予一个单调递增的整数，这个整数会附在该节点的路径后面。

在许多算法中，这有两个用处。首先，它提供了创建唯一节点的机制。这样的计数器能够确保节点名称不会重复。其次，在请求父节点的子节点时，它可以作为排序机制，用来实现领导者选举等算法。

3.3.1.3　版本号

znode在创建的时候都会被赋予一个版本号。只要节点关联的数据有变化，这个版本号就会增加。作为可选项，它可以被传给delete和setData操作，这样客户端可以确保没有意外重写对节点的修改。

3.3.2　监视和通知

事件等待是ZooKeeper的主要用例之一。这里的事件主要是指znode或znode的子节点中的数据发生改变。ZooKeeper不使用轮询方式，而是使用通知（notification）系统将变化通知客户端。

这里的通知也称为监视（watch）。应用要向ZooKeeper注册，才可以获悉特定znode路径上的变化。这是一次性操作，因此当通知被触发后，客户端必须注册监视来接收关于变更的通知。

持续监控某个路径时，有可能丢失通知。这可能发生在客户端接收到通知之后路径发生变化，但还没有对下一次通知设置监视时。还好ZooKeeper的设计者对这种情况早有准备，他们设置了一个也能从znode中读取数据的监视。这种有效方法使得客户端可以对通知进行合并。

3.3.3　保持一致性

ZooKeeper并不直接使用Paxos的一致性算法，而是使用称为Zab的另一种替代算法，Zab主要目的是提供比Paxos算法更好的性能。

当有变化发生时，Paxos算法会更新整个状态，与之不同的是，Zab算法会根据状态变化的顺序来运行。不过，Zab的许多特性还是与Paxos相同：领导者节点的选举需要经过提案阶段，并在接收到仲裁的接收者发回的确认之后，才能提交提案。Zab算法也使用与Paxos类似的计数器系统，称为时间戳（epoch number）。

3.3.4　创建ZooKeeper集群

ZooKeeper的设想是作为众多不同的应用的公共资源，为其提供共享服务。它使用仲裁的机器来维护高可用性。使用上一节提到的Zab算法，其中的一台机器会被选为领导者，然后所有变化都会被复制到仲裁机器中的其他服务器中。这一节讨论ZooKeeper服务器的安装，用来在单机开发环境或多服务器集群中使用。

3.3.4.1 安装ZooKeeper服务器

Hadoop发布版的安装包通常包含ZooKeeper。这些发布版往往包含兼容各种服务器风格的资源库。

也可以从Apache的ZooKeeper网站（http://zookeeper.apache.org ）下载二进制安装包来安装。在撰写本书时，ZooKeeper服务器的最新版本是3.4.5。一般来说，最新版本对大多数应用都是支持的，但与3.3系列服务器有所不同。对于需要3.3系列的软件来说，3.3.6是最新的稳定版本。

将ZooKeeper安装包解压之后，其目录结构应该如下所示：

```
$ cd zookeeper-3.4.5/
$ ls
CHANGES.txt              docs
LICENSE.txt              ivy.xml
NOTICE.txt               ivysettings.xml
README.txt               lib
README_packaging.txt     recipes
Bin                      src
build.xml                zookeeper-3.4.5.jar
conf                     zookeeper-3.4.5.jar.asc
contrib                  zookeeper-3.4.5.jar.md5
dist-maven               zookeeper-3.4.5.jar.sha1
```

在可以启动服务器之前，必须对其进行配置。在conf目录中有一个配置样例，对于新手进行ZooKeeper服务器的配置很有帮助。首先将这个文

件复制到zoo.cfg中，这是服务器启动脚本要调用的配置文件。其中前几个选项是用来处理心跳报文的时间间隔的，通常不用修改：

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
```

下一个参数是ZooKeeper数据目录的位置。默认情况下是/tmp/zookeeper，但最好马上修改它，因为多数系统会在一段时间之后丢弃/tmp目录中的数据。在这个例子中将其设置为/data/zookeeper。不管在什么情况下，这个目录以及其中的所有文件都应该由运行ZooKeeper服务器的用户所有，并对该用户是可写的：

```
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/data/zookeeper
```

尽管ZooKeeper的数据库不能超过RAM的可用空间，但你要确保数据目录中有足够的可用空间。ZooKeeper的数据库快照以及记录快照之间变化的日志都存放在这个目录里。

最后，要指定客户端及其他服务器与本服务器通信的端口：

```
# the port at which the clients will connect
clientPort=2181
```

除非碰巧与已有的其他服务相冲突，否则不要修改这个端口。

完成配置之后，在适当的位置（例如/data/zookeeper）创建数据目

录。在该目录中创建名为myid的文件，这个文件包含一个整数，这个整数对每个服务器都是唯一的。该文件用来将本服务器与集群中的其他服务器区别开来。没有它，ZooKeeper无法运行。对于仲裁中的每个节点，该整数的值都应在1～255：

```
echo 1 > /data/zookeeper/myid
```

以上操作都完成之后，就可以使用Zookeeper的bin子目录下的zkServer.sh脚本启动ZooKeeper守护进程。zkServer.sh脚本类似于init.d脚本，提供常见的停止、启动和重启命令。它还有前台启动（start-foreground）选项，这在开发时很有用。下列命令将守护进程的日志发送到前台而不是写入日志文件：

```
$ bin/zkServer.sh start-foreground
JMX enabled by default
Using config: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:QuorumPeerConfig@101] - Reading configuration
  from: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:DatadirCleanupManager@78] -
  autopurge.snapRetainCount
[myid:] - INFO  [main:DatadirCleanupManager@79] -
  set to 0
[myid:] - INFO  [main:DatadirCleanupManager@101] - Purge task is not
  autopurge.purgeInterval scheduled.
[myid:] - WARN  [main:QuorumPeerMain@113] - Either no config or no
  quorum defined in config, running  in standalone mode
[myid:] - INFO  [main:QuorumPeerConfig@101] - Reading configuration
  from: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:ZooKeeperServerMain@95] - Starting server
[myid:] - INFO  [main:Environment@100] - Server
  environment:zookeeper.version=3.4.5-1392090,
  built on 09/30/2012 17:52 GMT
[myid:] - INFO  [main:Environment@100] - Server
  environment:host.name=byrons-air
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.version=1.7.0_45
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.vendor=Oracle Corporation
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.home=/Library/Java/JavaVirtualMachines/
  jdk1.7.0_45.jdk/Contents/Home/jre
```

3.3.4.2　选择仲裁数量

在开发阶段时，单台ZooKeeper服务器通常是足够的。但是对于生产系统来说，使用单台ZooKeeper服务器会引入单点故障问题。要克服这个问题，就要部署一个称为仲裁的服务器集群，使其具备在单台机器发生故障时的容错能力。

ZooKeeper一个常被忽视的重要问题是，集群大小对于性能有着很大影响。由于要保持一致性，随着集群规模的增大，ZooKeeper用来改变状态所需要的时间也会随之增加。因此，尽管更多的服务器会提高容错能力，但容错和性能之间存在反相关的关系。

一般来说，不要让ZooKeeper集群超过5个节点，并且由于偶数个服务器不会提高容错能力，因此没必要小于3个节点。将集群规模定为3或5个节点是通常做法。如果应用对集群利用率较低，例如集群用于领导者选举或主要用来读取配置信息，那么可以用5个节点来提供更多的容错能力。如果集群利用率很高，比如用于像Kafka这样的应用（在第4章中讨论），那么3个节点所带来的性能提升更合适。

3.3.4.3　监控服务器

ZooKeeper提供了两种机制来监控仲裁的服务器组件。第一个机制是一个简单的监控关键字集合，它可以通过ZooKeeper指定的端口来访问。另一个机制是通过Java管理扩展（Java Management Extensions，JMX）接口。尽管这两种机制有一定的重叠，但它们有一些数据是互补的，因此在同一个环境下它们都是有用的。

ZooKeeper本地监控关键字俗称“四字真言”，它们是一系列命令，用

来返回ZooKeeper集群的状态信息。顾名思义，这些命令都由4个字母组成：dump、envi、reqs、ruok、srst和stat。这些命令会以明文传输到ZooKeeper的客户端端口，然后由ZooKeeper返回纯文本的输出信息。例如，要从shell命令行中发送ruok命令，只需要使用echo和netcat工具：

```
echo ruok | nc 127.0.0.1 2181
```

如果ZooKeeper服务器运行正常，服务器会返回imok响应。如果服务器的状态有问题，那就不会发送响应。通过周期性地发送这条命令，监控软件就能够确保所有节点都已启动并正常运行。

dump命令返回当前连接的会话，以及这些会话拥有的临时节点的相关数据。它必须在ZooKeeper集群当前的领导者机器上运行。在下面这个特殊例子中，运行着本章后面会讲到的分布式队列生产者示例。输入dump命令，会返回下列信息：

```
$ echo dump | nc 127.0.0.1 2181
SessionTracker dump:
Session Sets (2):
0 expire at Wed Jan 15 22:13:03 PST 2014:
1 expire at Wed Jan 15 22:13:06 PST 2014:

0x143999dfb5c0004
ephemeral nodes dump:
Sessions with Ephemerals (0):
```

在这个例子中，没有正在使用的临时节点，打开的会话是两个。

envi命令将信息从服务器的环境变量转储到标准输出。例如，假设ZooKeeper实例包含在Kafka 2.8.0（第4章介绍）中，在OS X系统上的Oracle Java 7环境下运行：

```
$ echo envi | nc 127.0.0.1 2181
Environment:
zookeeper.version=3.3.3-1203054, built on 11/17/2011 05:47 GMT
host.name=byrons-air
java.version=1.7.0_45
java.vendor=Oracle Corporation
java.home=/Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/
  Home/jre
java.class.path=:
     ../core/target/scala-2.8.0/*.jar:
     ../perf/target/scala-2.8.0/kafka*.jar:
     ../libs/jopt-simple-3.2.jar:
     ../libs/log4j-1.2.15.jar:
     ../libs/metrics-annotation-2.2.0.jar:
     ../libs/metrics-core-2.2.0.jar:
     ../libs/scala-compiler.jar:
     ../libs/scala-library.jar:
     ../libs/slf4j-api-1.7.2.jar:
     ../libs/slf4j-simple-1.6.4.jar:
     ../libs/snappy-java-1.0.4.1.jar:

     ../libs/zkclient-0.3.jar:
     ../libs/zookeeper-3.3.4.jar:
     ../kafka_2.8.0-0.8.0.jar
java.library.path=/usr/local/mysql/lib::
  /Users/bellis/Library/Java/Extensions:
  /Library/Java/Extensions:
  /Network/Library/Java/Extensions:
  /System/Library/Java/Extensions:/usr/lib/java:.
java.io.tmpdir=/var/folders/5x/0h_qw77n4q73ncv4l16697l80000gn/T/
java.compiler=<NA>
os.name=Mac OS X
os.arch=x86_64
os.version=10.9.1
user.name=bellis
user.home=/Users/bellis
user.dir=/Users/bellis/Projects/kafka_2.8.0-0.8.0
```

stat命令返回ZooKeeper服务器的性能信息：

```
$ echo stat | nc 127.0.0.1 2181
Zookeeper version: 3.3.3-1203054, built on 11/17/2011 05:47 GMT
Clients:
 /127.0.0.1:58503[1](queued=0,recved=731,sent=731)
 /127.0.0.1:58583[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/2/285
Received: 762
Sent: 761
Outstanding: 0
Zxid: 0x2f9
Mode: standalone
Node count: 752
```

如上述代码所示，stat命令返回ZooKeeper服务器的响应延迟、接收命令数量和发送响应数量的相关信息。它还报告服务器的模式，在本例中，这个模式是以开发为目的，搭配Kafka使用的standalone服务器。发送srst命令可以重置延迟和命令统计信息。在这个例子中，我们运行了一个队列示例，并在先前的stat命令之后发送srst。这就重置了计数统计量，但队列一直有新的元素加入，因而节点计数会不断增加：

```
$ echo srst | nc 127.0.0.1 2181
Server stats reset.
$ echo stat | nc 127.0.0.1 2181
Zookeeper version: 3.3.3-1203054, built on 11/17/2011 05:47 GMT
Clients:
 /127.0.0.1:58725[0](queued=0,recved=1,sent=0)
 /127.0.0.1:58503[1](queued=0,recved=1933,sent=1933)

Latency min/avg/max: 2/2/3
Received: 15
Sent: 15
Outstanding: 0
Zxid: 0x7ab
Mode: standalone
Node count: 1954
```

最后，reqs命令返回当前未完成的请求，这对于确定客户端是否阻塞很有用。

3.3.5 ZooKeeper本地Java客户端

ZooKeeper的软件库自带一个Java客户端，它可以在项目中使用。本节介绍怎样建立Maven项目来使用ZooKeeper软件库。另外还会介绍客户端的基本使用方法。

3.3.5.1 将ZooKeeper加入Maven项目

基本的ZooKeeper客户端软件库（撰写本书时的版本号是3.4.5）位于Maven的中心资源库（Maven Central），可以通过向项目的pom.xml中添加下列依赖，将其包含到项目中：

```
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.4.5</version>
  <exclusions>
    <exclusion>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
```

注意，我们刚开始将log4j工件（artifact）从ZooKeeper工件中排除，后来又手动将其加入。这解决了ZooKeeper 1.2.15中包含的Log4J版本一个特有的问题，它引用无法访问的库。如果发布了新版本的ZooKeeper，该问题可能会得到解决，那就不需要这种变通方法。

3.3.5.2 连接ZooKeeper

建立ZooKeeper会话是通过创建ZooKeeper对象完成的，对象的创建要使用标准构造器。最复杂的构造器如下所示：

```
public ZooKeeper(
  String connectString,
  int sessionTimeout,
  Watcher watcher,
  long sessionId,
  byte[] sessionPasswd,
  boolean canBeReadOnly
) throws IOException
```

最简短的构造器只需要connectionString和sessionTimeout参数。

connectionString参数是主机和端口号对的列表，以逗号间隔。在3.4及后续版本中，可以在主机名后附加一个可选的路径。在连接服务器时，这个路径的作用类似于chroot命令，能够告诉客户端将该路径作为服务器的根目录。

客户端在连接ZooKeeper时，首先将连接字符串中所列的主机打乱，然后尝试连接乱序列表中的第一个主机。如果连接失败，那就尝试第二个主机，以此类推。

例如，要连接zookeeper1、zookeeper2以及带有子目录other的zookeeper3这三台服务器，ZooKeeper客户端将初始化如下：

```
new ZooKeeper(
  "zookeeper1,zookeeper2,zookeeper3/other",
  3000,
  new Watcher() {
    public void process(WatchedEvent event) {
    }
  }
);
```

在这个例子中，代码中的Watcher接口是空实现，但常用来响应

ZooKeeper发送的所有通知。有时event对象的类型字段是空的，这时通知属于连接状态。其他时候，Watcher用来接收这样的监视事件：监视请求在设置监视时，没有指定具体Watcher对象。

ZooKeeper客户端使用的命令与上一节中的相同：create、delete、exists、getData和getChildren。这些命令都会立即返回结果。可以选择在特定路径上设置监视，或者将事件返回给先前指定的客户端层的监视，或者返回给传递给命令的其他Watcher对象。要想实际地了解，最容易的方式就是举例说明。

使用ZooKeeper选举领导者机器

ZooKeeper一个最常见的用处就是管理仲裁服务器，其管理方式与管理自身几无二致。在这些仲裁服务器中，其中一台要作为领导者，其他机器则或者用作副本，或者用作热备份，这取决于具体应用。

本例用最简单的算法来实现领导者选举：

·在目标路径上创建一个临时的顺序节点。

·检查子节点列表。其中最小的节点就是领导者。

·否则，监视子节点列表，一旦有改变就尝试再次选举。

这里，LeaderElection类实现了基于锁的领导者选举。首先用路径和ZooKeeper客户端来对类进行初始化：

```java
public class LeaderElection implements Watcher, ChildrenCallback {
  String    path;
  ZooKeeper client;
  String    node;
public Integer   lock       = new Integer(0);
public boolean   leader     = false;
public boolean   connected = false;

public LeaderElection(String connect,String path)
  throws IOException {
  this.path = path;
  client = new ZooKeeper(connect, 3000, this);
}
```

这个类将自己注册为对该客户端连接事件的Watcher。当客户端连接ZooKeeper时，它要请求得到领导队列中的一个位置。如果连接失败，它就放弃领导权（如果已经获得了领导权的话），并通知服务器连接已经丢失。具体处理由process方法实现：

```java
public void process(WatchedEvent event) {
  if(event.getType() == Watcher.Event.EventType.None) {
  switch(event.getState()) {
  case SyncConnected:
    connected = true;
    try {
      requestLeadership();
    } catch(Exception e) {
      //Quit
      synchronized(lock) {
        leader = false;
        lock.notify();
      }
    }
    break;
  case Expired:
  case Disconnected:
    synchronized(lock) {
      connected = false;
      leader    = false;
      lock.notify();
    }
    break;
  default:
    break;
  }
  } else {
    if(path.equals(event.getPath()))
      client.getChildren(path, true, this, null);
  }
}
```

在Watcher方法中，首先检查事件，看它是不是ZooKeeper状态事件（EventType.Node）。如果是连接事件，那就请求一个领导位置。如果不是连接事件，就检查路径。如果该路径与当初创建对象时注册的路径相同，那就获取子节点列表。这里，使用了回调事件，因此在检查领导状态时只需要检查一个位置。当调用getChildren时，还要重置监视。

请求领导权时，如果主路径不存在，那就首先创建主路径。这个调用可能会出现错误，因为每个客户端都会尝试创建该路径。如果在客户端调用create方法时该路径已经存在，那就可以放心地忽略这种错误信息。然后就在该路径上创建临时顺序节点。该节点是服务器的节点，是为所有将来的事务而存储的。在节点创建成功之前，对象将自己注册，以获知主路径的变化，这样当只有一个服务器启动时，新节点的创建行为会注册一个通知。这些都是在requestLeadership中实现的，如下所示：

```
public void requestLeadership() throws Exception {
  Stat stat = client.exists(path, false);
  if(stat == null) {
    try {
      client.create(path, new byte[0],

        ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
    } catch(KeeperException.NodeExistsException e) { }
  }
  client.getChildren(path, true, this, null);
  node = client.create(path+"/n_", new byte[0],
    ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
  System.out.println("My node is: "+node);
}
```

当getChildren调用返回结果时，使用了在ChildrenCallback接口中定义的一个回调方法processResult。在本例中，该方法以子节点数组作为参数调用isLeader，来查看它是否是子节点列表中的第一个节点。如果是的

话，就更新其领导者状态，并将领导者状态可能已经改变的消息通知给等待获取领导权的所有线程：

```
public void processResult(int rc,
    String path, Object ctx,
    List<String> children) {
  synchronized(lock) {
    leader = isLeader(children);
    System.out.println(node+" leader? "+leader);
    lock.notify();
  }
}
```

ZooKeeper在返回子节点列表时，并不保证列表元素是有序的。由于领导者被定义为具有最小序号的节点，因此要找出节点是否是领导者，首先就要对节点列表进行排序。然后检查第一个元素的节点名称：

```
protected boolean isLeader(List<String> children) {
  if(children.size() == 0) return false;
  Collections.sort(children);
  System.out.println(path+"/"+children.get(0)
    +" should become leader. I am "+node);
  return (node.equals(path+"/"+children.get(0)));
}
```

注意，getChilderen的响应并没有包含完整路径。应该在比较时添加完整路径，因为create方法不返回完整路径。

等待成为领导者的服务器，在循环中不带参数调用isLeader方法。如果该节点目前是领导者，该方法很快返回true，服务器就可以继续处理。如果不是领导者，主调线程就在lock对象处等待，直到获得了processResult的通知，或者服务器丢失与ZooKeeper的连接：

```
public boolean isLeader() {
  if(leader && connected) {
    System.out.println(node+" is the leader.");
    return leader;
  }
  System.out.println("Waiting for a change in the lock.");
  synchronized(lock) {

    try {
      lock.wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  return leader && connected;
}
```

为了查看实际运行的例子，假设有一个服务器。该服务器在处于领导者角色时，通告它正在“处理”循环中的某些事务，除此之外什么都不做。但是，这里有一定的不可靠性，它有30%的可能会崩溃，从而不得不放弃领导权。如果它没有崩溃但丢掉了领导权，那它就回去等待领导权。如果连接丢失，那它完全退出：

```java
public class UnreliableServer implements Runnable {

  String name;
  LeaderElection election;
  public UnreliableServer(String name,String connect,String path)
    throws IOException {
    this.name = name;
    election = new LeaderElection(connect,path);
  }

  public void run() {
    System.out.println("Starting "+name);
    Random  rng  = new Random();
    do {
      if(election.isLeader()) {
        System.out.println(name+": I am becoming the leader.");
        do {
          if(rng.nextDouble() < 0.30) {
            System.out.println(name+": I'm about to fail!");
            try {
              election.close();
            } catch (InterruptedException e) { }
            return;
          }
          System.out.println(name
            +": I'm leader and handling a work element");
          try {
            Thread.sleep(rng.nextInt(1000));
          } catch (InterruptedException e) { }
      } while(election.isLeader());
      //If we lose leadership but are still connected, try again.
        if(election.connected) {
          try {
            election.requestLeadership();
          } catch (Exception e) {
            return;
          }


        }
      }
    } while(election.connected);
  }

  }
```

要查看实际效果，那就用三台服务器搭建一个小测试，每台服务器都争夺领导权，直到崩溃为止：

```java
public class LeaderElectionTest {

  @Test
  public void test() throws Exception {
    Thread one   = new Thread(new UnreliableServer("huey","localhost"
      ,"/ducks"));
    Thread two   = new Thread(new UnreliableServer("dewey","localhost"
      ,"/ducks"));
    Thread three = new Thread(new UnreliableServer("louis","localhost"
      ,"/ducks"));

    one.start();
    two.start();
    three.start();

    //Wait for all the servers to finally die
    one.join();
    two.join();
    three.join();
  }
}
```

下面就是这个过程的输出示例。每次子节点列表有变化，这三台服务器都会检查它们是否是领导者。是领导者的服务器就开始处理，直到“崩溃”为止。这种情况一直持续到三台服务器全部崩溃：

```
Starting huey
Starting dewey
Waiting for a change in the lock.
Waiting for a change in the lock.
Starting louis
Waiting for a change in the lock.
My node is: /ducks/n_0000000009
My node is: /ducks/n_0000000010
My node is: /ducks/n_0000000008
/ducks/n_0000000009 leader? false
/ducks/n_0000000010 leader? false
Waiting for a change in the lock.
Waiting for a change in the lock.
/ducks/n_0000000008 leader? false
Waiting for a change in the lock.
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000008
```

```
/ducks/n_0000000008 leader? true
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000009
louis: I am becoming the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? false
/ducks/n_0000000009 leader? false
Waiting for a change in the lock.
Waiting for a change in the lock.
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm about to fail!
/ducks/n_0000000009 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? false
/ducks/n_0000000009 should become leader. I am /ducks/n_0000000009
/ducks/n_0000000009 leader? true
Waiting for a change in the lock.
dewey: I am becoming the leader.
dewey: I'm leader and handling a work element
/ducks/n_0000000009 is the leader.
dewey: I'm leader and handling a work element
/ducks/n_0000000009 is the leader.
dewey: I'm about to fail!
/ducks/n_0000000010 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? true
huey: I am becoming the leader.
huey: I'm leader and handling a work element
/ducks/n_0000000010 is the leader.
huey: I'm leader and handling a work element
/ducks/n_0000000010 is the leader.
huey: I'm about to fail!
```

对这个过程有一些优化措施，能够减少消息的发送量。具体来说，等待领导权的服务器实际上只需要监视上一个集合（而不是整个集合）的下一个最小节点。在该节点被删除后，就重复检查，来寻找最小节点。

3.3.6　Curator客户端

目前流行的Curator客户端是在ZooKeeper的本地Java API上进一步包装得来的。通常不直接使用ZooKeeper API，转而使用Curator客户端，因为它能够处理ZooKeeper的一些较为复杂的极端情况。具体来说，它可以解决ZooKeeper客户端中与连接管理相关的复杂问题。

Curator客户端最初是由Netflix开发的，后来贡献给了Apache基金

会，2013年3月开始孵化过程（http://curator.apache.org ）。

3.3.6.1　将Curator加入Maven项目

Curator软件库已经被分为多个不同的组件，它们都可以从Maven中央资源库中找到。将curator-recipes工件添加到项目中就可以将所有相关组件包含在内：

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>2.3.0</version>
</dependency>
```

多数用户只需要使用这个工件，因为它还包含了上一节所提供的recipes的深入实现。如果不包含recipes，通过curator-framework工件，该框架可以自行导入：

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>2.3.0</version>
</dependency>
```

3.3.6.2　连接ZooKeeper

Curator ZooKeeper客户端类是CuratorFramework，它是通过调用CuratorFramework-Factory类的newClient静态方法生成的。例如，连接一个本地ZooKeeper实例的代码如下所示：

```
CuratorFramework client = CuratorFrameworkFactory.newClient(
  "localhost:2181",
  new ExponentialBackoffRetry(1000,3)
);
```

第一个参数是常用的ZooKeeper连接字符串。照例，它可以包含一个以逗号间隔的可用主机列表，选择其中一个主机进行连接。第二个参数是重试策略，这是Curator独有的。上述例子中使用的是ExponentialBackoffRetry策略，超时时间为1000毫秒，最多重试3次。在该策略下，连接重试之间的等待时间会随着重试次数的增加越来越长。

Curator还有若干其他策略可以使用。RetryNTime策略与ExponentialBackoff策略类似，只是它的连接尝试之间的时间间隔是相同的，而不是不断增加。RetryUntilElapsed策略则将重试时间间隔设为固定量，这与RetryNTime策略相似。所不同的是，它会持续尝试连接直到过去了固定的时间，而后者则是尝试固定次数。

CuratorFrameworkFactory类也可以不立即创建客户端，而是返回一个连接的Builder类。这在用到Curator的某些高级特性时很有用。如果改用Builder方法进行连接，上述连接代码可以改写成：

```
CuratorFrameworkFactory.Builder builder =
  CuratorFrameworkFactory.builder()
    .connectString("localhost:2181")
    .retryPolicy(new ExponentialBackoffRetry(1000,3));
```

这里的代码也包含了基本的连接字符串和重试策略。在发生网络分区问题时，还可以通过调用canBeReadOnly方法使连接进入只读状态：

```
builder.canBeReadOnly(true);
```

可以使用namespace方法设置名字空间，它位于所有路径的前面：

```
builder.namespace("subspace");
```

最后，要创建CuratorFramework对象，就要调用build方法：

```
CuratorFramework client = builder.build();
```

CuratorFramework对象创建之后，必须用start方法开启连接。如果连接没有开启，多数CuratorFramework方法都无法使用。

start方法用来使应用可以绑定Listener方法，在事件到来之前监听客户端。当监听连接状态事件时，这特别重要。例如，下列代码片段来自于wiley.streaming.curator.ConnectTest，它将状态变化打印到控制台：

```
CuratorFramework client =
  CuratorFrameworkFactory.builder()
    .connectString("localhost:2181")
    .retryPolicy(new ExponentialBackoffRetry(1000,3))
    .namespace("subspace")
    .build();
client.getConnectionStateListenable().addListener(
  new ConnectionStateListener() {
    public void stateChanged(
      CuratorFramework arg0,
      ConnectionState arg1
    ) {
      System.out.println("State Changed: "+arg1);
      }
});
client.start();
Thread.sleep(1000);
```

在这个例子中，首先用Builder接口以及与之关联的名字空间来创建客户端。然后，实现了一个Listener来打印状态发生的变化。最后，用start启动客户端。如果在start命令后面加入一个简短的Thread.sleep命令，给客户端一点时间来连接，运行这段代码，就会输出已连接（CONNECTED）状态：

```
log4j:WARN No appenders could be found for logger
  (org.apache.curator.framework.imps.CuratorFrameworkImpl).
log4j:WARN Please initialize the log4j system properly.
State Changed: CONNECTED
```

3.3.6.3 使用znode

Curator框架的多数命令使用“流畅”风格。在这种风格的框架中，多数命令返回的对象可以被串联调用。上一节中的Builder对象就是这种接口的一个例子，Curator所有的znode操作例程也都是这样的。

使用流畅风格的Curator，直接支持将所有基本的znode操作表示为以forPath调用结尾的一连串方法调用。ForPath方法以一个完整路径以及一个可选的byte数组（这个数组包含与znode相关联的任意数据载荷）作为参数。下面这个例子以同步的方式创建一个新的路径，并根据需要创建父节点znode：

```
client.create()
  .creatingParentsIfNeeded()
  .forPath("/a/test/path");
```

通过向方法调用串中加入inBackground调用，这些方法也可以异步执行：

```
client.create()
.creatingParentsIfNeeded()
.inBackground(new BackgroundCallback() {

public void processResult(CuratorFramework arg0, CuratorEvent arg1)
  throws Exception {
    System.out.println("Path Created: "+arg1);
  }

})
.forPath("/a/test/path");
```

注意，inBackground方法返回PathAndBytesable＜T＞对象，而不是最初的builder。该对象只支持forPath方法，因此必须处于调用串的末尾。如果运行，参考wiley.streaming.curator.PathTest提供的例子，该命令应该有下列输出：

```
Path Created: CuratorEventImpl{
  type=CREATE,
  resultCode=0,
  path='/a/test/path',
  name='/a/test/path',
  children=null,
  context=null,
  stat=null,
  data=null,
  watchedEvent=null,
  aclList=null
}
```

如果要以另外的模式创建路径，那就在create命令中使用withMode方法。例如，队列系统可能想要创建既持久又顺序的znode：

```
client.create()
  .withMode(CreateMode.PERSISTENT_SEQUENTIAL)
  .forPath("/queue/job")
;
```

有效的模式包括PERSISTENT、EPHEMERAL、PERSISTENT_SEQUENTIAL和EPHEM-ERAL_SEQUENTIAL。也可以使用CreateMode.fromFlag方法来确定模式，正如wiley.streaming.curator.ModeTest例子中所演示的，这个方法将ZooKeeper本地客户端使用的标记（flag）转换为相应的CreateMode选项。

CheckExists命令有两个用处。首先，正如给定的其名字所暗示的那

样，可以用该命令是否返回null来检查路径是否存在。如果给定的路径已经存在，就会返回一个Stat对象，这个对象包含了关于该路径的丰富的有用信息，其中包括该路径的子节点数量、版本号，以及创建和修改的次数。

Znode的数据元素和子节点分别使用getData和getChildren命令来获取。与create命令类似，这两个命令可以在后台运行，并且总是以forPath方法结尾。与create命令不同的是，它们的路径中没有一个可选的字节数组。

直接使用GetData命令时，它会返回一个字节数组，这个数组可以反序列化为应用所需要的任何格式。例如，如果数据是简单的文本字符串，可以用如下方式对其设置然后获取（假设znode已经存在）：

```
client.setData()
  .forPath("test/string", "A Test String".getBytes());

System.out.println(
  new String(
    client.getData()
      .forPath("test/string")
  )
);
```

与create命令类似，getData可以在后台运行。在后台运行时，forPath方法返回null，数据则返回给后台回调函数。例如，以后台回调函数的形式实现上述代码，将会是如下形式：

```
client.getData().inBackground(new BackgroundCallback() {

public void processResult(CuratorFramework arg0, CuratorEvent arg1)
  throws Exception {
    System.out.println(new String(arg1.getData()));
  }

}).forPath("test/string");
```

上述两个例子都是在wiley.streaming.curator.DataTest中实现的。

GetChildren命令的工作方式与getData命令本质上是相同的。给定父对象的子znode节点的名称，它不是返回一个字节数组，而是返回一个String对象列表。下面的例子展示了该命令的使用效果，首先我们向一个假设已经存在的队列znode中添加几个任务，然后调用该节点的getChildren来获取子节点列表，详见wiley.streaming.curator.QueueTest：

```
if(client.checkExists().forPath("/queue") == null)
  client.create().forPath("/queue");

for(int i=0;i<10;i++) {
  client.create()
    .withMode(CreateMode.PERSISTENT_SEQUENTIAL)
    .forPath("/queue/job-");
}

for(String job : client.getChildren().forPath("/queue")) {
  System.out.println("Job: "+job);
}
```

运行上述代码，输出结果将是：

```
Job: job-0000000003
Job: job-0000000002
Job: job-0000000001
Job: job-0000000000
Job: job-0000000007
Job: job-0000000006
Job: job-0000000005
Job: job-0000000004
Job: job-0000000009
Job: job-0000000008
```

注意到Curator对getChildren调用并没有做什么特殊的工作，因此返回的子节点也是乱序的，这与ZooKeeper本地客户端一样。如果要实现FIFO（先进、先出）队列，必须对返回的列表进行排序，之后才能进行处理。

Stat对象也可以与其他从ZooKeeper中读取数据的命令联合使用，例如使用stroreStatIn方法的getData命令和getChildren命令。该方法以一个Stat对象作为输入，该对象的数据将由命令在运行时注入。例如，像上一个例子一样，从/queue中获取子节点列表的代码如下：

```
Stat stat = new Stat();
List<String> jobs = client.getChildren()
  .storingStatIn(stat)
  .forPath("/queue");
  for(String job : jobs)
    System.out.println("Job: "+job);
System.out.println("Number of children: "
  +stat.getNumChildren());
```

要删除路径，可以使用delete命令。与ZooKeeper本地客户端类似，可以使用withVersion命令将该命令设置为在特定版本上运行。下列例子查看路径是否存在，如果存在就将该特定版本删除，并且还会将znode拥有的所有子节点都删除：

```
Stat stat;
if((stat = client.checkExists().forPath("/a/test/path")) != null)
  client.delete()
    .deletingChildrenIfNeeded()
    .withVersion(stat.getVersion())
    .forPath("/a/test/path");
```

3.3.6.4 在Curator中使用监视

CheckExist、getData和getChildren命令都支持通过usingWatch方法对自己设置监视。该方法以一个对象作为输入参数，这个对象实现ZooKeeper本地的Watcher接口或针对Curator的CuratorWatcher接口。前面我们介绍过Watcher接口，CuratorWatcher接口本质上与之相同。例如，下列代码使用CuratorWatcher来观察特定路径的存在性是否有变化：

```
client.checkExists().usingWatcher(new CuratorWatcher() {

  public void process(WatchedEvent arg0) throws Exception {
    if (arg0.getType() == Watcher.Event.EventType.None) {
      //State change
    } else {
      System.out.println(arg0.getType());

      System.out.println(arg0.getPath());
    }
  }

}).forPath("/a/test/path");
```

3.3.7 Curator Recipes组件

除了实现健壮且易用的ZooKeeper客户端，Curator还进一步实现了ZooKeeper网站上的大量recipes。

本节给出的recipes基本上都是程序骨架（skeleton），主要作为常见情形下使用ZooKeeper的清晰示例。Curator recipes在基本骨架的基础上

提高了健壮性，增加了更丰富的特性，使其更适合在生产环境中使用。基于这些常见的算法结构，Curator recipes为我们实现高层逻辑提供了一个很好的起点。

本节涵盖了一些recipes的使用方法，包括了ZooKeeper网站的recipes部分中除了多版本并发控制之外的所有内容。其中某些情况下，例如对于分布式队列实现，还对基本算法进行了扩展，这样的扩展在ZooKeeper网站中是没有的。

3.3.7.1 分布式队列

Curator网站上的“Technical Note 4”文档声称，对于队列的实现来说，ZooKeeper是糟糕的系统。其中给出的原因包括，ZooKeeper的节点载荷比较小，它需要将整个数据集放在内存中，以及当znode有非常多的子节点时会有性能问题。

从某种意义上说，这种说法是完全正确的。在具有大流量消息的环境下进行队列处理，ZooKeeper是很差劲的选择。类似于ActiveMQ或RabbitMQ这样的专门的队列管理系统更合适。在流量非常大的环境下，下一章讨论的数据移动系统是更好的选择。不过，对于像作业队列这样的环境，其中工作节点分配到的项目相对较少，ZooKeeper是一个不错的轻量级解决方案。

在后者环境下，Curator提供了几个不同的分布式队列选项，它们都可以通过Queue-Builder对象来访问。举例来说，考虑一个需要处理WorkUnit对象的简单分布式队列。WorkUnit类是一个简单的例子，包含

用来处理某些任务所需要的信息。在这个例子中，它只包含一个标识符和一个载荷字符串：

```
public class WorkUnit implements Serializable {

  private static final long serialVersionUID = -2481441654256813101L;
  long    workId;
  String payload;

  public WorkUnit() { }
  public WorkUnit(long workId,String payload) {
    this.workId = workId;
    this.payload = payload;
  }

  public String toString() {
    return "WorkUnit<"+workId+","+payload+">";
  }

}
```

为了定义WorkUnit对象的生产者，首先定义DistributedQueue对象，并创建Curator客户端。这里，我们用连接字符串和只重试一次的策略定义该客户端。然后就启动客户端，开始处理ZooKeeper消息：

```
public class DistributedQueueProducer implements Runnable {

  DistributedQueue<WorkUnit> queue;

  public DistributedQueueProducer(String connectString)
    throws Exception {
    CuratorFramework client = CuratorFrameworkFactory.newClient(
      connectString,
      new RetryOneTime(10)
    );
    client.start();
```

接下来，就要创建队列自己了。QueueBuilder的builder方法总是以4个参数作为输入：Curator客户端、QueueConsumer＜T＞对象、QueueSerializer＜T＞对象、路径。这里，线程只扮演生产者角色，因此

QueueConsumer是null。然后，就创建和启动DistributedQueue对象。与客户端类似，队列只有启动之后才能使用：

```
queue = QueueBuilder.builder(client,
  null,
  new SimpleSerializer<WorkUnit>(),
  "/queues/work-unit")
.buildQueue();
queue.start();
```

Curator并不自带任何内置的QueueSerializer实现，而是完全交由应用为每个队列实现相应的序列化器（serializer）。这里，WorkUnit实现了Serializable的一个简单Java接口。序列化用来在WorkUnit对象和字节数组之间进行转换。下例的SimpleSerializer类实现了一个序列化器，这个序列化器可以用于实现Serializable的任何类：

```
blic class SimpleSerializer<T extends Serializable> implement
           QueueSerializer<T> {

blic byte[] serialize(T item) {
ByteArrayOutputStream bos = new ByteArrayOutputStream();
byte[] value = null;
try {
  ObjectOutput out = new ObjectOutputStream(bos);
  out.writeObject(item);
  value = bos.toByteArray();
  out.close();
} catch(IOException e) { }
try {
  bos.close();
} catch(IOException e) { }
  return value;
}

@SuppressWarnings("unchecked")
public T deserialize(byte[] bytes) {
```

```
  ByteArrayInputStream bin = new ByteArrayInputStream(bytes);
  Object object = null;
  try {
    ObjectInput in  = new ObjectInputStream(bin);
    object = in.readObject();
    in.close();
  } catch(IOException e) {
  } catch (ClassNotFoundException e) { }
  try {
    bin.close();
    } catch(IOException e2) { }
      return (T)object;
  }
}
```

现在，就只剩下向队列中添加元素这件事了。下面这个简单例子实现了Runnable接口，以及一个简单的运行循环，该循环每次等待不超过2秒的随机时间，然后向队列中添加元素：

```
Random rng = new Random();
public void run() {
  System.out.println("Starting Producer");
  boolean cont = true;
  long id = 0;
  while(cont) {
    try {
      queue.put(new WorkUnit(id++,"Next Work Unit "+id));
      System.out.println("Added item");
    } catch (Exception e1) {
      e1.printStackTrace();
    }
    try {
      Thread.sleep(rng.nextInt(1000));
    } catch (InterruptedException e) {
      cont = false;
    }
  }
}
```

接下来就该实现QueueConsumer了。这里，为了与生产者保持对称，QueueConsumer的实现中也有创建DistributedQueue的代码。因此，初看起来它与生产者的实现非常相似：

```
public class DistributedQueueConsumer
  implements QueueConsumer<WorkUnit> {
  DistributedQueue<WorkUnit> queue;
  String name;

  public DistributedQueueConsumer(
    String connectString,
    String name) throws Exception {

    this.name = name;
    CuratorFramework client = CuratorFrameworkFactory.newClient(
      connectString,
      new RetryOneTime(10)
    );
    client.start();

  queue = QueueBuilder.builder(
    client,
    this,
    new SimpleSerializer<WorkUnit>(),
    "/queues/work-unit"
  )
  .buildQueue();
  queue.start();
}
```

唯一的区别在于，消费者向QueueConsumer参数传递的不是null，而是它自己。它也实现了QueueConsumer接口，该接口需要实现两个方法。这里，对队列中元素的“处理”仅仅是将WorkUnit对象中包含的信息输出来而已：

```
public void stateChanged(CuratorFramework arg0, ConnectionState arg1) {
  System.out.println(arg1);
}

public void consumeMessage(WorkUnit message) throws Exception {
  System.out.println(name
    +" consumed "+message.workId+"/"+message.payload);
}
```

最后，可以创建下面这个简单进程，启动两个队列消费者来处理队列元素，以及一个队列生产者来生成新元素：

```
DistributedQueueConsumer q1 = new DistributedQueueConsumer(
  "localhost","queue 1"
);
DistributedQueueConsumer q2 = new DistributedQueueConsumer(
  "localhost","queue 2"
);
new Thread(new DistributedQueueProducer("localhost")).run()
```

开始运行后，该进程首先将队列中上次运行没有消费完的旧元素全部消费掉，然后再开始生产和消费新元素。注意，这些元素指定给哪个队列消费者是任意的，这取决于ZooKeeper服务器所触发的监视：

```
Starting Producer
Added item
queue 2 consumed 0/Next Work Unit 1
Added item
queue 2 consumed 1/Next Work Unit 2
Added item
queue 2 consumed 2/Next Work Unit 3
Added item
queue 1 consumed 3/Next Work Unit 4
Added item
queue 2 consumed 4/Next Work Unit 5
Added item
queue 2 consumed 5/Next Work Unit 6
Added item
queue 1 consumed 6/Next Work Unit 7
Added item
queue 1 consumed 7/Next Work Unit 8

Added item
queue 2 consumed 8/Next Work Unit 9
Added item
queue 2 consumed 9/Next Work Unit 10
Added item
queue 2 consumed 10/Next Work Unit 11
Added item
queue 1 consumed 11/Next Work Unit 12
```

3.3.7.2 选举领导者节点

Curator提供两种recipes来实现领导者的选举：LeaderSelector类和LeaderLatch类。Leader-Selector类使用回调机制来实现其功能，

LeaderLatch类则使用一个进程，该进程与先前的领导者选举例子很类似，该例子使用的是本地Java API。

为了使用LeaderSelector，尝试获取进程领导权的服务器必须实现LeaderSelectorListener接口。该接口指定了takeLeadership方法，该方法是服务器的主方法。从这个方法返回就会放弃领导权。例如，假设存在某个不可靠主机，如下所示：

```
public class UnreliableLeader implements LeaderSelectorListener {

  String  name   = "";
  boolean leader = false;
  String  connect;

  public UnreliableLeader(String connect,String name) {
    this.connect = connect;
    this.name = name;
  }


  public void stateChanged(CuratorFramework arg0,
    ConnectionState arg1) {
    if(arg1 != ConnectionState.CONNECTED) leader = false;
  }

  public void takeLeadership(CuratorFramework client) throws Exception {
    leader = true;
    Random rng = new Random();
    while(leader) {
      if(rng.nextDouble() < 0.30) {
        System.out.println(name+": crashing");
        return;
      }
      System.out.println(name+": processing event");
      Thread.sleep(rng.nextInt(1000));
    }
  }

}
```

LeaderLatch客户端采取另外一种方法，它提供了hasLeadership方法和await方法，前者用来检查服务器是否仍然有领导权，后者用来等待领导权。LeaderLatch的使用方法如下所示：

```
LeaderLatch latch = new LeaderLatch(client,"/leader_queue");
latch.start();
latch.await();
while(latch.hasLeadership()) {
  //Do work here
}
```

Wiley.streaming.curator.LeaderSelectionTest使用UnreliableLeader类来启动3个不同的领导者，这3个领导者一旦有一个发生故障，其他两个就会接手故障者的事件处理工作：

```
Thread[] thread = new Thread[3];
for(int i=0;i<thread.length;i++) {
  UnreliableLeader l = new UnreliableLeader(
    "localhost","server"+i);
  thread[i] = new Thread(l);
  thread[i].run();
}
for(int i=0;i<thread.length;i++) {
  thread[i].join();
}
```

这个例子的输出如下所示：

```
server0 is about to start waiting for leadership
server0: processing event
server0: processing event
server0: processing event
server0: processing event
server0: crashing
server1 is about to start waiting for leadership
server1: crashing
server2 is about to start waiting for leadership
server2: processing event
server2: processing event
server2: processing event
server2: crashing
```

3.4 总结

本章介绍了在分布式进程间维护共享状态以及进行协调背后的若干概念。多年以来，许多分布式系统都是用临时机制来处理这些任务，多数借助于关系数据库。关系数据库通常管用，但当要实现各进程各行其是（do-it-yourself）的分布式锁管理器等系统时，就有可能引入错误。

为了解决这个问题，Yahoo！等公司开发了协调工具，这些工具是作为服务发挥作用，而不是某个软件的组件。这催生了ZooKeeper的开发，并使其最终开源成为Apache项目。这种途径显然取得了成功：本书使用的绝大多数分布式软件都使用ZooKeeper进行配置和协调工作。虽然还有其他系统，例如还处于开发阶段的etcd，但到目前为止，ZooKeeper仍然是最流行的工具。

在接下来的几章中，我们通过讨论分布式系统来将ZooKeeper投入实际应用。ZooKeeper的最大用户是Kafka项目，该项目是第4章讨论的一个数据移动系统。ZooKeeper还频繁用于流处理应用的协调任务，例如Storm和Samza，它们都使用了ZooKeeper项目。

ZooKeeper在终端用户的应用中也很有用。它可以帮助管理数据模式等分布式状态。此外，它还可以根据需要，对有限的共享资源提供访问控制功能。正是因为它的通用性和有效性，我们才在本章不惜篇幅详细讨论。

第4章　流分析中的数据流程管理

在第3章中，我们介绍了维护分布式状态的概念和其中存在的困难。之所以要维护分布式状态，其中一个最常见的原因，就是需要以可扩展的方式来采集和处理数据。

分布式数据流程的管理出现已久，它涉及数据的处理和采集两种任务。一般来说，处理这种任务的系统一直是内部开发或是外包开发的定制应用。直到最近几年，用来实现这类系统的技术才发展到了可以建立公共基础架构的地步。与协调和配置系统类似，数据流程系统也可以划分出来成为一个独立的服务。现在，它们在接口和假设方面已经足够通用，能够跳出最初针对的应用独立使用。

这类系统中，最早的大概是队列系统，例如21世纪初面世的ActiveMQ。但是，队列系统实际上并不是为高吞吐量应用设计的（尽管许多系统现在可以有很好的性能），并且往往是以Java中心。

后来就出现了由Facebook这样的大型互联网公司开源出来的系统。2008年发布的Scribe工具就是这一代系统中最著名的一个。该工具使用类似RPC的机制，将数据从边缘服务器集中到像Hadoop这样的处理框架中。Scribe有着许多与当今一代系统相同的特性，包括将数据脱机保存到磁盘等能力，但它只能处理间歇性的连接失败。

由Cloudera开发的Flume，以及kafka，都是当今一代的分布式数据采集系统，它们分别代表了两种截然不同的理念。对于这两种数据移动系统

的维护和支持，本章都进行了讨论。另外，还讨论了它们当初的设计理念，以帮助我们决定在何种情形应该采用何种系统。不管怎样，这两种数据移动系统不应被视为是互斥的。根据需要，它们完全可以共存于一个环境中。

4.1 分布式数据流程

分布式数据流程系统应该具备两个基本属性："至少一次"的交付语义和"n + 1"交付问题。没有这两个属性，分布式数据流程就很难成功扩展。本节涵盖了这部分内容，并讨论了这两个属性对分布式数据流程如此重要的原因。

4.1.1 至少交付一次

在任何类型的数据采集框架中，数据交付和处理都有三个选择：

·最多交付一次

·至少交付一次

·恰好交付一次

许多交付框架，特别是系统监控所使用的框架，提供的是"最多一次"的交付和处理语义。多半因为这些框架在当初设计时，所针对的情形并不需要传送全部数据，但是需要有最高的性能，从而在发生问题时及时向管理员发出预警。实际上，许多这类系统会对数据进行降频采样，从而进一步提高性能。只要大概知道数据丢失率，监控软件就能够在处理时恢复有用的值。

在用日志计费的金融或广告等其他系统中，每条数据记录的丢失都意味着收益损失。而且，审计的需要往往意味着无法通过监控领域所用的技

术来估计数据丢失情况。这种情况下，多数实现方案转而通过队列系统实现“恰好交付一次”交付语义。使用这种语义的流行的例子包括Apache的ActiveMQ队列系统、RabbitMQ，以及数不清的商业解决方案。这些服务器通常在服务器端实现队列语义，这主要是因为它们的设计目标是支持企业环境下的各种数据生产者和消费者。

不管是否需要，“恰好交付一次”交付机制都提供了交付的安全性，与“最多一次”交付机制性比，这当然是以牺牲性能为代价的。“至少一次”交付系统尝试在这两种极端机制之间寻找平衡点，它将处理语义推给消费者来提供可靠消息交付。这样，消费者就可以使用交付的数据流来创建应用所需要的语义，而不影响数据流的其他消费者。例如，需要进行“恰好交付一次”处理（比如两个账户间的金融交易）的消费者，可以自行实现去重过程，来确保消息没有被重复处理。另一个只关心数据跟踪（例如不同账户交易对的数量）的消费者就不需要“恰好交付一次”交付语义，因为它使用幂等集（idempotent set）操作，这意味着它不需要付出管理代价。一般认为，除了确保至少交付一次，消息应该怎样处理取决于应用逻辑，应该在应用逻辑层面解决。

4.1.2 “n + 1”问题

在“传统”日志处理系统中，架构基本上都是“漏斗”形态。来自于大量边缘系统的数据被采集到少数几个中心地点（通常是一个，但由于法律或物理上的限制，有时会需要多个处理地点）。然后这些处理地点就对这些数据进行操作，有时会将它们移动到数据仓库等其他地点进行深入分析。

第一个“漏斗”处理完之后，下面一系列事情就不可避免：首先，要加入第二个数据消费者，因而还要再创建一个“漏斗”。接着，可能还要引入其他前端服务，这些服务都有自己的数据采集机制。最终，每当加入了新的服务或处理机制，就必须与其他所有系统集成起来。这看似夸张，实际上却是工业界十分常见的反模式。

企业界是用面向服务架构（Service-Oriented Architecture，SOA）的形式来解决这个问题的，进而发展了企业服务总线（Enterprise Service Bus，BUS）的思想。该思想的主要内容是，对于不使用公共协议各自独立实现的服务，由总线处理它们之间的交互，并将它们之间的通信标准化。在实际情况中，总线常负责在企业内部的不同“孤岛”之间移动数据，例如，在销售企业的Saleforce实现和分析团队的内部数据库之间移动数据。

对本章讨论的数据流程工具来说，情况则恰恰相反。这里的总线层和每个应用之间的通信都是标准化的，消息系统主要负责管理系统间的物理流程。这样，任意数量的生产者和消费者都可以通过数据流程协议的公共机制来通信，而不用管其他生产者和消费者。

4.2 Apache Kafka：高吞吐量分布式消息机制

Apache Kafka项目是由LinkedIn开发的，本来用于连接其网站服务和内部的数据仓库基础架构。2011年LinkedIn将其开源，发布了0.6号版本。当年不久Kafka就被Apache接受为孵化器项目。Kafka目前发布的是0.8号版本，这个版本加入了许多新特性，包括新的生产者消息API和内部复制功能。本节讨论的是0.8号版本系列，不过很多企业往往依然使用更简单的0.7.2版本，因为这个版本的第三方客户端更丰富。

4.2.1 设计与实现

Kafka是为了解决特定公司的特定问题专门创建的，这里所说的公司当然是LinkedIn，所说的问题就是大流量消息以及大量不同服务间的互相通信。截至2013年年末，不包括数据中心之间的镜像数据消息，LinkedIn自称每天处理近600亿条消息。该公司还集成有大量服务，这些服务很可能包含数百个不同的生产者和消费者。LinkedIn基本上能够完全掌握自己的环境和软件，因此能够创建专有性很强的数据移动环境。

这种环境并不是LinkedIn独有的。许多主要与“互联网数据”打交道的公司也处于与之相同的环境。这些公司许多都是工程型公司，这意味着多数软件是他们自行开发，而不是从第三方购买的。这样他们就可以使用Kafka模型或者Amazon.com最近公布的类似系统Kinesis。Kinesis的目标是构建核心组件将Amazon.com的各种服务集成起来，这些服务包括键-值存储Dynamo、块存储引擎S3、Hadoop基础架构Elastic Mapreduce，以及

高性能数据仓库Redshift。

本节主要介绍Kafka的内部设计，以及它们是怎样集成起来解决前面讲到的分布式数据流程问题的。

4.2.1.1　主题、分区和代理

“主题”（topic）是Kafka的组成要素。在Kafka系统中，主题是一个数据物理分区。包含在主题中的所有数据都具有一定的联系。最常见的情况是，主题中消息的联系仅仅是，它们都可以由同一个公共机制来解析。

主题可以进一步细分为许多分区（partition）。分区的数量实际上限制了I/O受限的消费者从Kafka中提取数据的速度。这是因为客户端对每个分区往往只使用一个消费者线程（或进程）。例如，如果用Camus工具（该工具使用Hadoop把数据从Kafka移动到Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）），一个（Mapper）就可以从多个分区中提取数据，而多个Mapper不会从同一个分区中提取数据。

分区也常用来在逻辑上组织一个主题。生产者的实现通常会提供机制，基于给定消息的键值来为消息选择Kafka分区。

分区自身是分散存在于代理（broker）之中，代理是组成Kafka集群的物理进程。集群中的每个代理通常对应一个独立的物理服务器，并管理向该服务器磁盘的所有写操作。然后这些分区就会被均匀分散到各个代理中。在Kafka 0.8和后续版本中，还会将分区副本分散到集群中的其他代理中。

如果代理的数量发生变化，分区和分区的副本可以被重新指定给集群中的其他代理。如果要从主题中消费数据，消费者应用或者消费者组（如果应用是分布式的）会为每个分区指定一个线程或进程。随后，与Map-Reduce应用的Map阶段非常相似，这些独立的线程按自己的节奏处理所负责的分区。Kafka消费者的高层实现跟踪各线程消费情况，如果某个进程或线程发生中断，就将处理重启。尽管可以绕过这个模型直接使用Kafka的底层接口，从而提高数据的消费能力，但首选机制还是在必要时增加主题的分区数量。为此，Kafka提供了工具向实际运行环境中的已有主题加入分区。

4.2.1.2 结构化的日志存储

Kafka基于只追加（append-only）的日志机制，这个机制类似于数据库应用的预写日志（write-ahead-log）协议。据说该协议最早是由Ron Obermark于1974年开发的，当时他还在IBM从事Sytem R的研究工作。从本质上说，预写日志是在对象可以被变更之前，就必须首先将变更提交到恢复日志中。这就为数据库的变更或删除操作生成了“撤销”（undo）日志，从而能在任何时候及时地重建数据库状态。

为了达到这个目标，构造恢复日志时，每次变更时都会设置一个唯一的递增的数字（数学上称为严格单调递增序列），将其附加到文件（理论上是无限大的）末尾。这个递增的数字一般是该记录在文件中的位置，在将记录附加到文件末尾时，通常很容易得到这个位置。当然，在实际系统中，没有什么文件是无限大的，因此当文件大小达到上限时，就要创建一个新文件，并重新设置偏移量。如果该文件也是用严格单调递增序列命名

的，那么就完全能够保持预写日志的语义。

这种方式不但简单，还能让保存这些日志的存储介质的性能最大化。在预写日志刚开发出来的时候，需要把日志写到磁带上（每盘磁带大约存储50MB的数据）。磁带当然最适合顺序写入。即使是现代的旋转存储介质（硬盘），其顺序读写性能也通常要比随机读写高很多。

从本质上说，Kafka使用的也是预写日志协议，它为系统中每个主题的每个分区维护单独的日志。当然，Kafka并不是数据库。它不是修改数据库中的记录，而是在将消息的变更提交给日志后，才向消费者提供这条消息，如图4-1所示。这样，没有哪个消费者会消费可能在代理故障事件中丢失的消息。

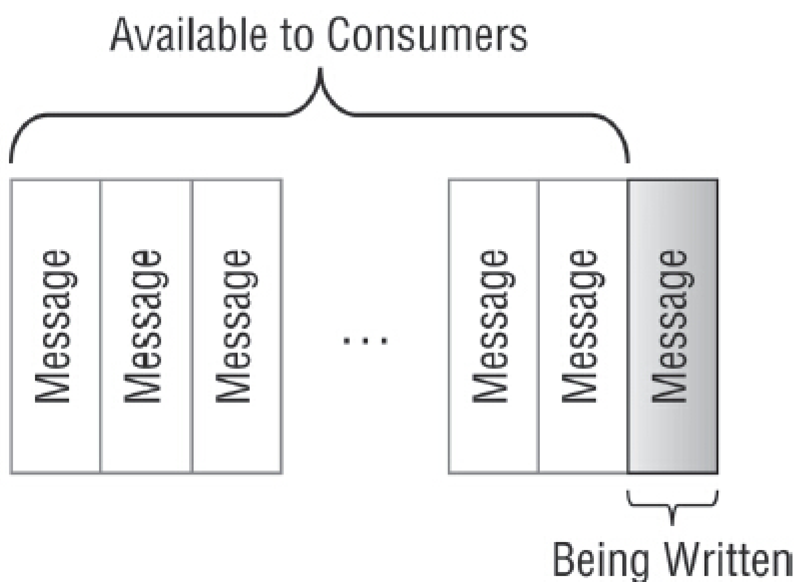


图　4-1

在LinkedIn的开发人员博客上，Kafka的开发人员Jay Kreps随手评论道：Kafka实现了“日志即服务”。他的话应该毫不夸张。实质上，Kafaka确实可以用来实现对传统关系数据库管理系统（Relational DataBase Management System）的数据复制系统。为此，要使用标准的生产者协议将对数据库表的修改写到Kafka中。数据库自身则是Kafka数据的消费者（在0.8版本中可以使用生产者回调函数来完成，此时应将生产者的响应设置为“所有”副本），并通过读Kafka来完成表的修改。为了减少恢复时间，Kafka会不定期地将表的快照以及每个分区最后的偏移量保存到磁盘。似乎Kafka的开发人员也在考虑这类应用。https://cwiki.apache.org/confluence/display/KAFKA/Log+Compaction 上的日志压缩提案给出了这样一种环境，其中的主题是长期存在的，并允许消费者恢复给定键的最近值。在数据库应用中，这涵盖了恢复和复制用例。数据库应用中的键是表的主键，消费者应用则负责修改本地拷贝，使其成为更适合查询的格式（例如B树）。

4.2.1.3　存储空间管理

尽管磁盘空间比较便宜，但它既不是免费的，也不是无限的。这意味着日志最终必将会从系统中被删除。Kafka与许多其他系统不同，它使用基于时间的日志保留机制。这就是说，除非特别说明，这些通常打包成1GB文件的日志都会在保留规定的几小时之后被删除。

这意味着Kafka能够使用它的整个磁盘，并拥有与0.7.x系列相同的故障模式。Kafka会持续地接受消息，不管是否能够成功地将消息保留在非易失性存储中。初看起来，这似乎是造成灾难的潜在因素。但是，管理可

用的存储通常要比管理缓慢的客户端或对旧数据的请求要容易得多。存储空间管理通常是核心系统监控的内置组件，一旦发生问题，通过为每个代理加入更多磁盘或加入更多代理和分区，这很容易弥补。

4.2.1.4 非队列系统

显然，Kafka绝不是像ActiveMQ或RabbitMQ这样的队列系统，它不会像这些队列系统那样费尽周折来确保消息的处理顺序与消息的接收顺序一致。Kafka的分区系统不会维护这样的结构。它对于特定主题的分区的读写没有固定顺序，因此不保证客户端从分区中的读取顺序与写入顺序一致。另外，生产者实现中的异步性也并不少见，这就导致可能发生这样的情况：一条消息首先发送到一个分区，而由于延迟差异或者其他非确定性事件，直到后发生的另一条消息发送到其他分区，这条消息才被写入。

这通常没什么问题，因为在许多互联网应用中，保持显式的顺序没有太大必要。事件的顺序实质上刚开始就是任意的，因此在一种任意顺序上保持另一种特定的任意顺序，这不会带来实际上的好处。进一步说，顺序很重要的事件通常发生的时间间隔很远，因此看起来仍然是被处理系统排过序的。

在如何处理消息的消费者这一方面，Kafka也与许多队列系统不同。在许多队列系统中，消息被消费之后，就会从系统中删除。Kafka则没有这种删除消息的机制，而是依靠消费者来跟踪所消费的最后一条消息的偏移量。尽管Kafka自带的高层消费者通过使用ZooKeeper管理偏移量，从而对此有所简化，但代理自身不负责消费者或其状态。

4.2.1.5　复制

Kafka在0.8版本中为代理集群引入了真正的复制功能。在这之前，任何复制的概念都必须通过集群到集群的镜像特性来处理，我们在下一节介绍该特性。需要指出的是，该镜像特性只是一种备份策略，而不是复制策略。

Kafka是在主题层进行复制的。主题的每个分区都有一个首席分区（leader partition），这个分区被零个或多个追随分区（follower partition）所复制（在Kafka 0.8中，未复制的主题就是没有追随分区的首席分区）。这些追随分区散布在不同物理代理中，目的是使每个追随分区都位于不同的代理中。因此，实现方案需要保证，代理的数量对于主题要使用的分区和副本数量是足够的。

创建分区时，所有这些追随分区都会被视为是“同步”的，从而形成一个同步副本集合（In-Sync Replica set，ISR）。当消息到达要写入的首席分区时，它首先会被附加到首席分区的日志中。然后它会被转发给当前处于ISR中的所有追随分区。当ISR中的每个分区都确认收到了这条消息，就认为这条消息已经完成提交，可以由消费者读取。首席分区也会不定期地记录一个高水位标志（high watermark），这个标志包含了最新提交消息的偏移量，并将其传播给ISR中的追随分区。

如果包含个别副本的代理发生故障，那它会被从ISR中删除，首席分区不会等待它响应。Kafka使用Zookeeper来完成上述操作，从而始终维护一个“活跃”的代理集合。然后首席分区会继续使用ISR中缩小的副本池来处理消息。当这个副本恢复时，它检查自己最后知晓的高水位标志，并对

日志进行裁剪以适合该分区的大小，然后从当前位置开始到首席分区当前提交的偏移量为止，进行数据复制。这些工作完成后，就可以将该副本重新加入ISR，一切恢复如初。

向Kafka加入复制功能也向Kafka的生产者API引入了一些变化。在0.8版本之前的Kafka中，生产者API中没有确认功能。应用向Kafka socket写入数据，至于是否真正写入只能听天由命。在Kafka 0.8中，则有三个不同等级的确认功能：无确认、首席确认和全部确认。

第一个选项“无确认”与Kafka 0.7以及之前的版本相同，不会向生产者返回任何响应信息。这种情况允许数据丢失，是最不持久的，但性能最高，每秒能轻松处理数万条消息。

第二个选项“首席确认”在首席分区收到消息之后且从ISR收到确认消息之前发送确认。这降低了性能，并且仍然可能有数据丢失，但为多数应用提供了合理水平的持久性。

最后一个选项“全部确认”只有在首席分区提交消息之后才发送确认。这种情况下，只要ISR中至少还有一个分区，数据是不会丢失的。但是，相比“无确认”，这种情况下的性能损失是巨大的。不过，这些损失大多可以通过使用大量分区以及高并发的生产者实现方案来弥补。

4.2.1.6　多数据中心部署

许多Web应用是延迟敏感的，因此需要在全球范围内地理分布化。由于数据中心之间相距遥远，其网络连接可靠性必然要比数据中心内部的连接可靠性差。Kafka通过提供内置的镜像工具，来帮助处理数据中心之间

可能（令人沮丧的是，这是常见情况）增加的延迟以及网络连接完全中断问题。使用这些工具，在每个数据中心建立Kafka集群，并设计了一个保留时间，以在长时间停机和远程数据中心的可用空间之间进行平衡。如果有足够的可用空间，就可以有较长的保留时间作为灾备手段。

然后，就可以用MirrorMaker工具将这些远程集群复制到主处理集群中。Kafka自带的这个工具可以从多个远程集群中读取消息，并将消息写入单个输出集群。写入每个集群中同一个主题的消息会被合并到输出集群的单个主题中。

除了对远程数据中心进行镜像，镜像功能对于开发工作也很有用。可以简单地将远程集群镜像到开发集群，从而在开发过程使用生产数据，同时又不会向生产环境引入风险。

4.2.2　配置Kafka环境

这一节介绍怎样启动Kafka环境。在开发系统中，通常情况下用单个代理进行测试是足够的，然而，根据数据复制需求的不同，更大规模的系统很可能至少需要3到5个代理。高端系统则可能需要很多代理。第5章介绍了Samza工具，这个工具使用Kafka进行数据处理，其使用方式与Hadoop Map-Reduce使用分布式文件系统的方式相同。在这个例子中，集群中就可能有数十个运行Kafka的代理。

本节涵盖了安装和运行Kafka的全部内容。其中还有一个小节是关于在单台机器上使用多个独立配置进行多代理应用开发的。

4.2.2.1 安装Kafka

安装Kafka只需要将当前版本的Kafka解压并配置即可，写作本书时Kafka的版本号是0.8.0。Kafka自身主要是用Scala语言编写的，该语言有多个版本。为了避免版本之间的混淆，Kafka的开发人员将Scala的构建版本号包含在归档文件名中。这一点对于想要使用归档文件中包含的库进行开发的开发人员往往很重要。Kafka目前的二进制安装包是用Scala 2.8.0构建的，因此归档文件被命名为kafka_2.8.0-0.0.tar.gz。你可以从http://kafka.apache.org 下载该文件并解压：

```
$ tar xvfz ~/Downloads/kafka_2.8.0-0.8.0.tar.gz
x kafka_2.8.0-0.8.0/
x kafka_2.8.0-0.8.0/libs/
x kafka_2.8.0-0.8.0/libs/slf4j-api-1.7.2.jar
x kafka_2.8.0-0.8.0/libs/zkclient-0.3.jar
x kafka_2.8.0-0.8.0/libs/scala-compiler.jar
x kafka_2.8.0-0.8.0/libs/snappy-java-1.0.4.1.jar
x kafka_2.8.0-0.8.0/libs/metrics-core-2.2.0.jar
x kafka_2.8.0-0.8.0/libs/metrics-annotation-2.2.0.jar
x kafka_2.8.0-0.8.0/libs/log4j-1.2.15.jar
x kafka_2.8.0-0.8.0/libs/jopt-simple-3.2.jar
x kafka_2.8.0-0.8.0/libs/slf4j-simple-1.6.4.jar
x kafka_2.8.0-0.8.0/libs/scala-library.jar
x kafka_2.8.0-0.8.0/libs/zookeeper-3.3.4.jar
x kafka_2.8.0-0.8.0/bin/
[ More Output Omitted ]
$
```

多数操作系统将进程可以打开的文件数量限制为一个较小的值，例如1024。由于诸如主题、分区数量以及保留时间等多方面原因，Kafka可能需要消耗比常规操作更多的文件句柄。提高这个限制门槛的方法取决于各个操作系统，在Linux中要修改/etc/security/limits.conf文件。例如，加入下列的nofile（代表“文件数量（number of files）”而不是“没有文件（no files）”）行，就可以将允许kafka用户打开的文件数量提高到

50000：

```
kafka          -     nofile         50000
```

4.2.2.2 安装Kafka的前提

Kafka主要的外部依赖就是安装ZooKeeper。在生产环境中，请参考上一章中的安装指南来运行ZooKeeper集群。对于本地开发环境来说，Kafka自带了预配置的ZooKeeper服务器，可以用来在单个机器上运行多个代理。

为了启动开发版Kakfa，Kafka包含了脚本bin/zookeeper-server-start.sh和配置文件config/zookeeper.properties。你可以在终端窗口中启动服务器：

```
$ cd kafka_2.8.0-0.8.0/
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
INFO Reading configuration from:
  config/zookeeper.properties

  (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
WARN Either no config or no quorum defined
  in config, running  in standalone mode
  (org.apache.zookeeper.server.quorum.QuorumPeerMain)
INFO Reading configuration from:
  config/zookeeper.properties
  (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
INFO Starting server
  (org.apache.zookeeper.server.ZooKeeperServerMain)
[ Other Output Omitted ]
```

启动ZooKeeper之后，剩下事情的就是配置和启动Kafka代理。

4.2.2.3 配置代理

代理的名称与Kafka服务器相同。与ZooKeeper类似，代理也是用定义

代理本身的简单属性文件来配置的。关于主题和偏移量数据的信息保存在数据本身或ZooKeeper集群中。本节我们详细分析这个属性文件，对每一项设置及其对集群的影响加以解释。

在例行的Apache许可证信息之后，第一项设置就是代理标识符。Kafka将代理表示为唯一的无符号32位整数（内部编码为Java的long类型，但必须是正值，因此有效范围被限制为32位），不像ZooKeeper那样将id限制为0~255：

```
# The id of the broker. This must be set to a unique integer
# for each broker.
broker.id=0
```

如果使用IPV4地址，可以选择将IPV4地址转换为无符号32位数字作为唯一的代理id。许多编程语言有一个inet_aton函数，用来将“点分”形式的IP地址转换为无符号32位整数。也可以使用下列单行命令在Linux Bash shell上计算得到同样的结果，前提是已经安装了bc工具：

```
$ IP=$(ip=`hostname -I` && ip=( ${ip//\./ } ) \
>  && echo "(${ip[0]} * (2^24)) + (${ip[1]}*(2^16)) \
>  + (${ip[2]}*(2^8)) + ${ip[3]}" | bc) \
> && echo "broker.id: $IP"
```

上述代码使用Linux的hostname命令加上-I-选项获取“默认”的IP地址。ip =（${ip//\./}）将命令分解成IP地址的组成部分。注意“/”符号后面的空格很重要，不能省略。然后使用bc命令将这些组成部分乘上相应的数值，从而完成位移。

broker.id值只能在初始配置时设置，IP地址只能用来在集群启动时确保ID的唯一性。如果服务器由于重启而被分配新的IP地址，它的broker.id

还是应该保持原来的值。当然，当添加了新服务器，并且如果分配给该服务器的IP地址在集群初始配置期间已经被分配过，就可能出现问题。

下一个参数是代理用来通信的端口号。默认情况下是9092，除非要在同一台机器上运行多个代理实例（参见4.2.2.5节），通常不需要改变这个默认值：

```
# The port the socket server listens on
port=9092
```

host.name选项用来设置代理报告的主机名。多数情况下，这可以注释掉，因为getCanonicalHostName（）做了同样的事情。只有当计算环境比较独特，服务器有多个主机名时，才可能有必要显式地设置主机名。Amazon的EC2环境就是一个这样的例子。EC2中的每台服务器都有至少两个完全限定的域名（内部域和外部域），并且可以用Amazon的Route 53 DNS服务来赋予其他名字。默认主机名可能并不适于报告，这取决于哪个服务需要与集群通信。如果不能在操作系统层面解决这个问题，可以在这里显式地定义主机名，以使元数据调用返回正确的主机名。

```
# Hostname the broker will bind to and
# advertise to producers and consumers.
# If not set, the server will bind to all
# interfaces and advertise the value returned from
# from java.net.InetAddress.getCanonicalHostName()
#host.name=localhost
```

后面有两个选项来控制Kafka用于处理网络请求和管理日志文件的线程的数量。在旧版本的Kafka中，只有一个选项kafka.num.threads来设置线程数量。新的设置使得对Kafka集群的调优更具灵活性。num.network.threads的默认值是3，num.io.threads的默认值是8。对多数

用例来说，num.network.threads为三个线程通常足够了。对num.io.threads来说，推荐使用的线程数量应与用来保存日志的存储磁盘的数量相等。

```
# The number of threads handling network requests
num.network.threads=3

# The number of threads doing disk I/O
num.io.threads=8

# The number of requests that can be queued for I/O before network
# threads stop reading
#queued.max.requests=
```

接下来设置控制socket缓冲区的大小和socket请求的大小。在多数据中心传输的情况下，LinkedIn的参考配置从下面配置中的1MB提高到2MB。如果将这些设置注释掉，缓冲区大小的默认值就是较小的100KB，这对于海量的生产环境来说太小了。这里我们将请求大小的上限设为100MB，通常没必要对其修改。

```
# The send buffer (SO_SNDBUF) used by the socket server
socket.send.buffer.bytes=1048576

# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=1048576

# The maximum size of a request that the socket server
# will accept (protection against OOM)
socket.request.max.bytes=104857600
```

log.dir属性控制Kafka日志的输出位置。这些位置包括＜topic name＞-＜partition number＞形式的许多分区目录以及一个名为replication-offset-checkpoint的文件：

```
$ ls
analytics-0  audit-1  replication-offset-checkpoint
```

这个文本文件不能删除，它包含前面的4.2.1.5节中提到的“高水位标志”数据。如果Kafka收到多个以逗号间隔的目录，它会将分区在这些目录之间均匀分配。这是一种临时性选项，针对的是默认配置是JBDO（Just a Bunch of Disk）的服务器，例如Amazon EC2。不过可惜的是，Kafka只是随机地将分区分配给各磁盘，而不是将同一个主题的分区在多个磁盘间分配。主题的数据规模往往差异很大，有时不同磁盘的使用特征也会有很大差别。如果可能的话，使用RAID0将这些磁盘合并成一个大磁盘，从而使用一个日志目录，通常效果会更好。

```
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs
```

接下来，下列设置控制主题的创建和管理。

```
# Disable auto creation of topics
#auto.create.topics.enable=false

# The number of logical partitions per topic per server.
# More partitions allow greater parallelism
# for consumption, but also mean more files.
num.partitions=2
```

Kafka集群默认的复制因子是1，这意味着只有领导者机器才有数据。可以通过显式地创建主题来逐个主题地手工设置复制因子，不过所有自动创建的主题都会使用默认的复制因子（如果允许自动创建的话）。

```
# Set the default replication factor for a topic
#default.replication.factor=3
```

如果正在使用复制功能，那么下面几个参数负责控制ISR集合的维护。一般来说，这些参数的默认值多数没有必要修改。（如果实在要修改一定要倍加小心。）特别是，将最大延迟时间或最大消息数量设置得太小

会让副本无法留在ISR中。有一项设置有时可以提高性能，那就是加大num.replica.fetchers以提高复制吞吐量。

```
# The maximum time a leader will wait for a fetch request before
# evicting a follower from the ISR
#replica.lag.time.max.ms=10000

# The maximum number of messages a replica can lag the leader before it
# is evicted from the ISR
#replica.lab.max.messages=4000

# The socket timeout for replica requests to the leader
#replica.socket.timeout.ms=30000

# The replica socket receive buffer bytes
#replica.socket.receive.buffer.bytes=65536

# The maximum number of bytes to receive in each replica fetch request
#replica.fetch.max.bytes=1048576

# The maximum amount of time to wait for a response from the leader
# for a fetch request
#replica.fetch.wait.max.ms=500

# The minimum number of bytes to fetch from the leader during a request
#replica.fetch.min.bytes=1

# The number of threads used to process replica requests
#num.replica.fetchers=1

# The frequency with with the high watermark is flushed to disk
#replica.high.watermark.checkpoint.interval.ms=5000
```

下面一组设置控制Kafka的磁盘输入/输出（I/O）行为。当不使用复制来提高生产者和消费者的kafka吞吐量时，这些设置可以用来平衡数据持久性。

log.flush.interval.messages和log.flush.interval.ms这两个参数分别控制刷新事件的最小和最大间隔。log.flush.interval.ms是Kafka直到刷新（flushing）磁盘前的最长等待时间，因此它定义了刷新事件的上限。

log.flush.interval.messages参数定义Kafka收到多少消息之后才将消息刷新到磁盘。如果分区采集消息的时间快于刷新间隔时间，这个参数就定义了刷新率的下限。

如果没有使用复制，刷新事件的时间间隔就等价于代理故障事件中丢失的数据量。默认情况下是10000条消息或1000毫秒（1秒），对于大多数用例来说，这通常是足够的。在将这些值设置调高或调低之前，需要考虑两个方面的问题：吞吐量和延迟。磁盘刷新是Kafka最耗时的操作，因此提高刷新率会降低系统整体的吞吐量。反之，如果刷新的数据量很大，那就需要较长的时间。当有大块数据刷新到磁盘时，时间间隔的增加可能导致对生产者和消费者的响应延迟的激增。

可以使用log.flush.intervals.messages.per.topic和log.flush.intervals.ms.per.topic逐个主题地分别控制这两个参数。这些参数的取值是以逗号间隔的＜topic＞：＜value＞元组列表。

```
# The number of messages to accept before forcing
# a flush of data to disk
log.flush.interval.messages=10000

# The maximum amount of time a message can sit in a
# log before we force a flush
log.flush.interval.ms=1000

# Per-topic overrides for log.flush.interval.ms
#log.flush.intervals.ms.per.topic=topic1:1000, topic2:3000
```

下面一组参数控制主题的保留时间以及日志段的大小。日志总是以整段的方式删除的，因此段文件的大小以及保留时间共同决定了过期数据的删除速度。

控制保留时间的基本参数是log.retention.hours。当日志段的年龄超过该参数值时，就要被删除。删除操作是逐个分区进行的。还有一个参数是log.retention.bytes，但这个参数在实际中很少使用。这是因为，首先，前面我们讲过，基于空间上的限制来删除日志段要比基于时间更难管理。其次，log.retention.bytes使用32位整数而不是Java的64位长整型整数，这就将主题大小限制为每个主题每个分区最大2GB，这对开发和测试环境勉强可以，但很难满足生产环境的需要。

```
# The minimum age of a log file to be eligible for deletion
log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the

# log as long as the remaining segments don't drop
# below log.retention.bytes.
#log.retention.bytes=1073741824
```

每个段的大小是由log.segment.bytes和log.index.size.max.bytes控制的。前者控制日志段的大小，日志段是包含发送到Kafka的实际消息的文件。消息的磁盘格式与传输格式是相同的，这样消息就可以快速刷新到磁盘。文件的默认大小是1GB，在下面的例子中，为了开发目的将其设置为512MB。

Log.index.size.max.bytes参数控制对应每个日志段的索引文件的大小。该索引文件存储了关于文件中每条消息的偏移量的索引信息。默认情况下会用稀疏文件方法为该文件分配10MB空间，但是如果在段文件本身填满之前它就已经是满的，那就会导致新日志段结转（roll over）。

```
# The maximum size of a log segment file. When this size is
# reached a new log segment will be created.
log.segment.bytes=536870912
```

检查段是否过期的频率是由log.cleanup.interval.mins控制的。默认情况下是每分钟检查一次，通常情况下这个频率是足够的。

```
# The interval at which log segments are checked to see if they can
# be deleted according to the retention policies
log.cleanup.interval.mins=1
```

我们在4.2.2.2节曾经提到，Kafka使用ZooKeeper来管理给定集群中的代理的状态以及与之相关的元数据。Zookeeper.connect参数定义了管理代理的元数据的ZooKeeper集群。它是标准的ZooKeeper连接字符串，采取以逗号间隔的主机名的形式。它还允许代理通过指定连接字符串中的默认路径，来利用ZooKeeper的类似于chroot的行为。这在一个ZooKeeper集群中驻留有多个独立的Kafka集群时很有用。

```
# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify
# the root directory for all kafka znodes.
zookeeper.connect=localhost:2181
```

与其他ZooKeeper客户端类似，Kafka允许控制连接和会话的超时时间值，分别由zookeeper.connection.timeout.ms和zookeeper.session.timeout.ms参数来设置，默认情况下都是6000毫秒（6秒）。

```
# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=1000000

# Timeout in ms for the zookeeper session heartbeat.
#zookeeper.session.timeout.ms=6000
```

Kafka 0.8不但引入了复制，还引入了对代理的受控关闭行为。参数controlled.shutdown.enabled负责控制该行为，该参数默认情况下处于激

活状态。如果处于激活状态，当领导者节点收到关闭命令时，它首先尝试将所负责的每个主题分区的控制权重新指定给ISR中的其他代理，总共尝试controlled.shutdown.max.retries次，两次尝试的时间间隔以每次controlled.shutdown.retry.backoff.ms毫秒的速度递增。最后一次尝试之后它才关闭，这时关闭很可能是不完全的。

这个特性使得对集群的维护更加容易。假设消费者和生产者知道元数据的变化情况，那么将领导权交给其他代理之后，我们在整个集群升级的时候就感觉不到任何停机时间。可惜的是，多数客户端不具备良好的恢复功能，因此目前这并不可行。

```
#controlled.shutdown.enabled=true
#controlled.shutdown.max.retries=3
#controlled.shutdown.retry.backoff.ms=5000
```

本节介绍的多数参数都可以在Kafka安装目录的config/server.properties文件中找到。该属性文件通常用于开发环境。在生产环境中，除了socket缓冲区选项，多半可以使用默认设置。不管什么情况，都必须设置broker.id、log.dirs和zookeeper.connect属性。

4.2.2.4　启动Kafka代理集群

创建了合适的Kafka配置文件之后，就可以用kafka-server-start.sh脚本启动Kafka。该脚本以包含配置信息的属性文件作为输入：

```
$ ./bin/kafka-server-start.sh config/server.properties
```

服务器一般是在前台启动，因为它没有内置守护进程模式。在后台启动Kafka服务器的一个常见模式是，将输出写入日志文件和使服务器在后

台运行，例如init.d脚本：

```
$ ./bin/kafka-server-start.sh \
> config/server.propertiers > kafka.log 2>&1 &
```

4.2.2.5　单台机器上的多代理集群

有时候在开发阶段，需要有多个Kafka代理来确保应用能够正确处理多台服务器上的分区，并能管理对ISR的读取等。例如，为新语言开发客户端就应该始终在多代理集群上进行。

在单台机器上启动多个代理简单又直接。首先必须为每个代理指定一个单独的代理id、端口和日志目录。例如，可以使用下列三个文件：

```
config/server-1.properties:
broker.id=1
port=6001
log.dirs=/tmp/kafka-logs-1
zookeeper.connect=localhost:2181

config/server-2.properties:
broker.id=2
port=6002
log.dirs=/tmp/kafka-logs-2
zookeeper.connect=localhost:2181

config/server-3.properties:
broker.id=3
port=6003
log.dirs=/tmp/kafka-logs-3
zookeeper.connect=localhost:2181
```

启动各个代理与在单台服务器上启动一个代理是非常相似的。唯一不同的是要指定不同的Java管理扩展（Java Management Extension，JMX）端口。否则，就会由于第一个Kafka代理已经绑定了JMX端口而导致第二个和第三个代理无法启动。

4.2.3 与Kafka代理交互

可以从多种平台访问Kafka，但Kafka自带的是Java软件库，因为它本来就是在Java虚拟机（JVM）上运行的。它在本地客户端中，为写入Kafka的应用提供了一个选项，为从Kakfa读的应用提供了两个选项。

要在Java中使用Kafka，就必须在项目中包含Kafka软件库和Scala运行时库。可惜，由于Scala开发方式的原因，其运行时库通常是与构建时所基于的Scala软件库绑定的。对于纯Java项目，这没什么问题，但对于Scala项目这可能会带来麻烦。

下列例子使用纯Java语言。所使用的Kafka是基于Scala 2.8.0构建的：

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.8.1</artifactId>
  <version>0.8.1-SNAPSHOT</version>
</dependency>

<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.8.1</version>
</dependency>
```

4.2.3.1 生产者

要使用Kafka生产者，就要构造或从.properties文件中载入一个Properties对象。在生产系统中，Properties对象通常是从文件中载入的，这样不需要重新编译项目就可以对其修改。在下面这个简单例子中，Properties对象是构造得来的。随后这些Properties就被传递给

ProducerConfig对象的构造器：

```
public static void main(String[] args) throws Exception {

   Properties props = new Properties();
   props.put("metadata.broker.list",
     "localhost:6001,localhost:6002,localhost:6003");
   props.put("serializer.class", "kafka.serializer.StringEncoder");
   props.put("request.required.acks", "1");
   ProducerConfig config = new ProducerConfig(props);
```

接下来，该配置对象就被传递给生产者对象，生产者对象的类型由Key和Message的类型组成。Key通常是可选的，但很重要的一点是，配置属性中定义的serializer.class要能成功将Key和Message的对象编码。在这个例子中，Key和Message都是Strings类型：

```
String topic = args[0];
Producer<String,String> producer = new Producer<String,String>(config);
```

下面这个简单的例子从控制台读取用户输入，将其发送给Kafka。生产者可以发送单个KeyedMessage对象，也可以发送KeyedMessage对象列表。要发送的消息是用主题（topic）、键（key）和消息（message）构造的。如果不需要键，那就像这个例子一样只使用主题和消息：

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(System.in));
while(reader.ready()) {
  String toSend = reader.readLine();
  producer.send(new KeyedMessage<String,String>(topic, toSend));
}
```

4.2.3.2 消费者

对于Kafka消费者实现，应用有两种选择。一种选择是高层实现，使用ZooKeeper来管理和协调消费者组。另一种选择是低层实现，针对的是

需要管理自己的偏移量信息或者需要访问Simple Consumer中没有的特性的应用，这些特性包括将偏移量重置为流的起点。

下面这个例子使用Simple Consumer API将生产者产生的消息输出到控制台。与生产者的例子类似，首先要用Properties对象构造ConsumerConfig对象：

```
public static void main(String[] args) {
  Properties props = new Properties();
  props.put("zookeeper.connect", "localhost:2181");
  props.put("group.id", "console");
  props.put("zookeeper.session.timeout.ms", "500");
  props.put("zookeeper.sync.timeout.ms", "500");
  props.put("auto.commit.interval.ms", "500");
  ConsumerConfig config = new ConsumerConfig(props);
```

多数消费者应用程序是多线程的，其中许多是从多个主题中读取。为了达成这个目标，高层消费者实现一个主题映射，其内容为主题和应为每个主题创建的消息流数量。通常会将这些流分发到线程池进行处理，但在下面这个简单的例子中，我们只为主题创建了一个消息流：

```
String topic = args[0];
HashMap<String,Integer> topicMap = new HashMap<String,Integer>();
topicMap.put(topic,1);

ConsumerConnector consumer=Consumer.createJavaConsumerConnector(config);
Map<String,List<KafkaStream<byte[],byte[]>>> consumerMap =
  consumer.createMessageStreams(topicMap);
KafkaStream<byte[],byte[]> stream = consumerMap.get(topic).get(0);
```

KafkaStream对象定义了ConsumerIterator类型的迭代器，用来从流中提取数据。调用Iterator的next方法会返回一个Message对象，这个对象可以用来提取键和消息。

```
ConsumerIterator<byte[],byte[]> iter = stream.iterator();
while(iter.hasNext()) {
  String message = new String(iter.next().message());
  System.out.println(message);
}
```

4.3 Apache Flume：分布式日志采集系统

Kafka是在特定的生产环境中设计和构建的。这种情况下，其设计者实质上能够完全掌控其软件栈。此外，他们一直致力于将已有组件替换为类似的生产者/消费者语义（尽管交付语义不尽相同），这使得从工程角度上来说Kafka比较容易集成。

然而在许多生产者环境中，开发人员就没有这么幸运，因为经常会有大量已有的遗留系统、数据采集或日志记录路径。并且，由于各种各样的原因，我们无法修改这些系统的日志记录行为。例如，软件可能是由外部供应商生产的，有着自己的生产目的和开发路线图。

要与这样的环境集成，我们可以使用Apache Flume项目。Flume是对各种数据服务的数据入口和出口进行集成的框架，而不是像Kafka那样的一种独门的日志采集机制。最初，Flumes主要是从Web服务器这样的数据来源中采集日志，然后传给Hadoop环境中的HDFS中。后来，Flume逐步扩展，涵盖了更广泛的用例，有时甚至模糊了本章讨论的数据移动系统与第5章的处理系统之间的界线。

本节介绍Flume框架。该框架本身自带了许多开箱即用的组件，利用这些组件可以迅速搭建有用的基础架构。我们会对这些基本组件和用处详加介绍。也可以用Flume来开发定制组件，这通常是为了处理特定业务问题的数据格式或处理需求。Flume与Kafka类似，都是在JVM上开发的，因此对该框架进行扩展是相当简单的。

4.3.1 Flume agent

Flume主要的组织特征就是使用网络流模型（network-flow model）。它将该模型封装到一台机器中，这台机器称为agent。这些agent机器基本类似于Kafka的代理，一个agent仅由三个组件组成：一个source、一个channel和一个sink。一个agent包含的这类基本的流（flow）的数量是任意的，其组织形式如图4-2所示。

如图4-2所示，source是agent中流的起点。source组件从外部数据源中提取或接收数据。多数情况下，这些数据来自于应用。例如，Web服务器可以生成文件系统日志。要提取这些日志，将一个agent安装在运行Web服务器的机器上，agent中包含一个可以用tail命令处理日志文件的source。

source收到数据后，将数据传送给一个或多个channel。channel定义了持久语义和事务语义。根据所使用的channel的不同，agent可以将其数据保存到磁盘或数据库，也可以保留在内存中。这样每个agent可以选择不同的性能和可靠性特征。

source也可以选择将数据写到多个channel中，如图4-3所示。这种方式通常用于数据的多路复用，以提高负载均衡和并行处理能力。也可以用来进行数据复制或为整个系统提供多个数据路径。source使用选择器（selector）来定义数据写入多个channel的方式。默认情况下使用复制选择器，它将同一份的数据写到所有channel中。内置的另外一个选项是多路复用选择器，该选择器使用数据元素中的信息来确定到source的每段数

据的目的channel。

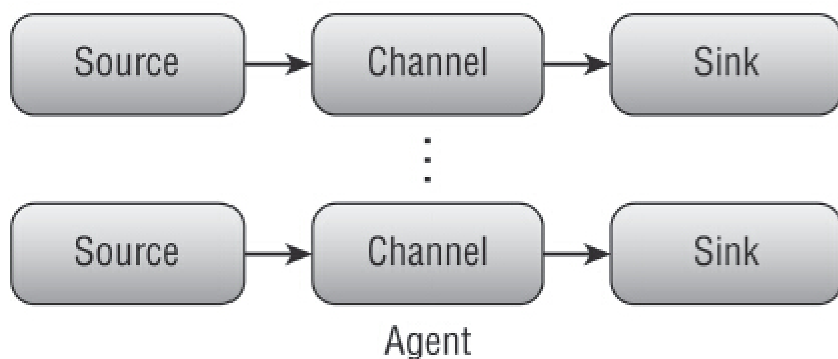


图　4-2

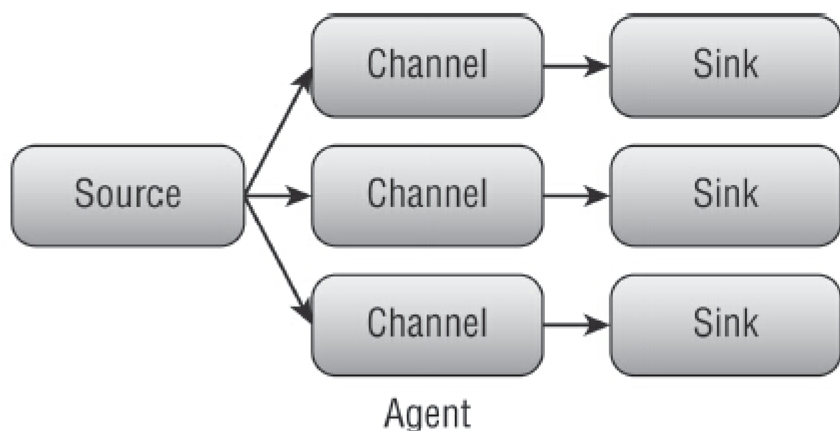


图　4-3

sink将数据从channel中删除，并将其重放到最终输出（例如HDFS）或其他agent。与source不同，每个sink都只关联一个channel。为了进行解复用（de-multiplexing），要将来自多个sink的输出发送给一个agent。

也可以用sink processor对多个Sink进行一定的协调。有三种内置的sink processor：默认processor、负载均衡processor和故障切换processor。在本章后面我们会对此详细讨论。

4.3.2 配置agent

配置agent的方法与Kafka很类似，也是使用属性文件。属性文件定义source、channel和sink的布局，以及个别元素的配置。该布局对于所有组合都是相同的，而个别配置则取决于每个元素的类型。对于机器上运行的每个agent，该配置首先定义source和sink，基本形式如下：

```
<agent name>.sources= <source1> ... <sourceN>
<agent name>.channels= <channel1> ... <channelN>
<agent name>.sinks= <sink1> ... <sinkN>
```

要用唯一的名字来替换掉＜agent name＞和＜source1＞，这样才能在属性文件的其他地方识别出这些元素。例如，定义有两个agent的属性文件，可以用如下方式：

```
agent_1.sources= source-1
agent_1.channels= channel-1 channel-2
agent_1.sinks= sink-1 sink-2

agent_2.sources= source-1
agent_2.channels= channel-1
agent_2.sinks= sink-1
```

在这个例子中，配置了两个agent，每个都有一个source。第一个agent名为agent_1，使用两个channel和两个sink来对输入进行复制或复用。第二个agent名为agent_2，它更简单，只有一个channel和一个sink。

注意，不同agent之间的source、channel和sink的名字不需要是唯一

的。配置信息始终只是针对一个agent定义而言的。

接下来，将每个agent的source和sink与对应的channel绑定起来。source是用＜agent name＞.sources.＜source＞.channels的形式与channel绑定的，sink则是用＜agent name＞.sinks.＜sink＞.channel的形式绑定的。在这个例子中，source绑定到所有channel，sink只与同名的channel绑定：

```
agent_1.sources.source-1.channels= channel-1 channel-2
agent_1.sinks.sink-1.channel= channel-1
agent_1.sinks.sink-2.channel= channel-2
agent_2.sources.source-1.channels= channel-1
agent_2.sinks.sink-1.channel= channel-1
```

4.3.3 Flume数据模型

在继续配置Flume agent之前，需要对Flume内部的数据模型进行一些讨论。该数据模型是通过Event数据结构定义的。Flume对该数据结构的接口定义如下：

```
package org.apache.flume;

import java.util.Map;

/**
 * Basic representation of a data object in Flume.
```

```
 * Provides access to data as it flows through the system.
 */
public interface Event {

  public Map<String, String> getHeaders();

  public void setHeaders(Map<String, String> headers);

  public byte[] getBody();

  public void setBody(byte[] body);

}
```

与Kafka的Message类似，Event载荷是称为Body的不透明字节数组。Event也允许引入头部（header），这与HTTP调用很类似。这些头部信息用于存放不属于消息本身的元数据。

头部通常是由Source引入的，其目的是加入与不透明事件相关的元数据。例如产生事件的机器的主机名或事件数据的数字签名。许多Interceptor也修改头部元数据。Flume使用元数据将事件发送到对应的目的地。例如，多路复用选择器使用头部来确定事件的目的channel。至于每个组件是怎样修改或使用元数据的，我们会在每个选择器的对应章节中详细讨论。

4.3.4　channel选择器

如果一个source使用多个channel，就像前面例子中的agent_1的source那样，那就必须使用某种策略在不同channel间分发数据。Flume的默认策略是使用复制选择器来控制数据分发，但它也内置有多路复用选择器，用来进行负载均衡或将输入划分给多个sink。另外，还可以实现channel选择器的定制行为。

4.3.4.1 复制选择器

复制选择器就是简单地将每个输入都发送到所有channel进一步处理。由于它是默认选择器，因此不需要进行配置，但也可以通过将selector.type属性设置为replicating来显式地定义：

```
agent_1.source.source-1.selector.type= replicating
```

对复制选择器只有一个额外选项，用来忽略向特定channel写入时发生的错误。将selector.optional属性设置为以空格间隔的channel列表，就可以将写入这些channel时发生的错误忽略掉。例如，下列设置使得agent_1的第二个channel成为可选的：

```
agent_1.source.source-1.selector.optional= channel-2
```

4.3.4.2 多路复用选择器

另外一个内置的channel选择器是多路复用选择器。该选择器用来根据Event的头部元数据决定怎样在不同channel中分发数据。要激活该选择器，就要将source的selector.type改为multiplexing：

```
agent_1.source.source-1.selector.type= multiplexing
```

默认情况下是根据元数据中的flume.selector.header头部来确定channel的，但这可以通过selector.header属性来修改。例如，下列代码将选择器改为使用较短的timezone头部：

```
agent_1.source.source-1.selector.header= timezone
```

多路复用选择器还需要知道怎样将来自于头部的值分配给绑定source

的各个channel。目标channel是用selector.mapping属性来指定的。必须显式地将每个可能的值映射到selector.default属性指定的默认channel。例如，要将Pacific Daylight Time（PDF）和Eastern Daylight Time（EDT）映射到channel-2，同时所有其他事件映射到channel-1，配置应该如下所示：

```
agent_1.source.source-1.selector.mapping.PDT= channel-2
agent_1.source.source-1.selector.mapping.EDT= channel-2
agent_1.source.source-1.selector.default= channel-1
```

多路复用选择器需要显式地指定头部到channel的映射。对于这一点是否会降低多路复用选择器的实用性是值得讨论的。实际上，这使得它更适合用作路由工具，而不是提高处理并行性的工具。如果更希望提高并行性，更好的选择是负载均衡sink处理器，本章随后对此加以讨论。

4.3.4.3　实现定制选择器

Flume也支持定制选择器的实现。要在配置中使用定制选择器，只需要为selector.type指定完全限定的类名（Fully Qualified Class Name，FQCN）。例如，下列代码告诉Flume对agent_1的source使用RandomSelector：

```
agent_1.source.source-1.selector.type=
  wiley.streaming.flume.RandomSelector
```

根据选择器实现的具体情况，还可能需要正确设置其他配置参数。选择器本身是通过继承AbstractChannelSelector类来实现的：

```
import org.apache.flume.Channel;
import org.apache.flume.Context;
import org.apache.flume.Event;
import org.apache.flume.channel.AbstractChannelSelector;

public class RandomChannelSelector extends AbstractChannelSelector {

  List<List<Channel>> outputs = new ArrayList<List<Channel>>();
  List<Channel>       empty   = new ArrayList<Channel>();
  Random              rng     = new Random();
```

这个类需要实现三个不同的方法。第一个是配置方法configure，该方法要提取指定给该source的channel的信息，以及在配置中设置过的其他所有属性。

```
public void configure(Context context) {
```

与context对象交互是非常简单的。context类定义了许多“getter”方法，可以用来提取配置参数。这些属性已经针对相应的选择器进行特化。例如，如果该类需要像复制选择器那样访问“optional”属性，下列代码就会获取该属性关联的字符串：

```
String optionalList = context.getString("optional");
```

不过，RandomSelector只需要与source相关的channel列表。GetAllChannel方法实际上是AbstractChannelSelector类的一部分，因此可以直接调用。由于RandomSelector将每个事件都发送给一个channel，因此构造元素列表供以后使用：

```
for(Channel c : getAllChannels()) {
  List<Channel> x = new ArrayList<Channel>();
  x.add(c);
  outputs.add(x);
}
```

定制channel选择器必须实现的其他两个方法是getRequiredChannels和getOptional-Channels。每个方法都只有一个Event类型的输入参数，该参数的元数据是可以查看的。RandomSelector不查看元数据，它只是返回一个随机数组，该数组包含一个在配置方法中构造的channel：

```
public List<Channel> getRequiredChannels(Event event) {
  return outputs.get(rng.nextInt(outputs.size()));
}

public List<Channel> getOptionalChannels(Event event) {
  return empty;
}
```

4.3.5 Flume source

Flume的一个主要吸引力就是它能够与各种系统集成。为了支持这一点，Flume自带了大量内置的source组件。它还支持定制source的实现，其实现方式与上一节讨论的channel选择器几无二致。

4.3.5.1 Avro source

Avro是Yahoo！开发的结构化的数据格式。它是Flume的本地数据交换格式，其理念与Google的protocol buffer和Facebook的Thrift等其他数据格式类似。这三个系统都使用接口定义语言（Interface Definition Language，IDL）来定义类似于C的structs或简单Java类的结构。根据IDL，可以用本地语言将定义好的结构编码成高性能的二进制有线传输协议的格式。通过访问IDL定义的模式，其他服务就可以很容易地将该格式解码。对于Avro而言，该IDL是JSON，因此易于定义可用多种语言实现的数据结构。

与Thrift类似，Avro还定义了一个远程过程调用（Remote Procedure Call，RPC）接口。这使得我们可以通过TCP/IP连接发送命令，来执行Avro定义的“方法”。Flume就是使用RPC接口来将事件传递给agent的。

在Flume中将type属性设为avro，就可以将该source设置为Avro source。Avro source还有两个属性bind和port，它们分别定义agent用于监听Avro RPC调用的网络接口和端口号。例如，下列选项将Avro listener绑定到系统在端口4141定义的所有网络接口。

```
agent_1.source.source-1.type=avro
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

使用这个特殊的IP地址0.0.0.0时要倍加小心。如果agent在拥有外部IP地址的服务器上运行，它就绑定到该IP地址。多数情况下，这是没有问题的，因为服务器几乎肯定处于防火墙的防护之下，能够免遭恶意攻击。但是，它也可能向不应有访问权限的内部服务开放端口。例如，生产服务器可能会无意间允许交付准备服务器（staging server）对其访问。如果该交付准备服务器被意外地错误配置，它就可能向系统注入不应有的数据。

除了type、bind和port属性，Avro source还有其他一些可选属性。例如，将compression.type设为deflate来压缩Avro数据流；还可以将ssl属性设置为true并将keystore属性指向对应的JavakeyStore文件，利用安全套接字协议层（Secure Socket Layer，SSL）将数据流加密。设置Keystore-password属性指定KeyStore文件自身的口令。最后，通过设置keystore-type来指定KeyStore类型，其默认值是JKS。

4.3.5.2　Thrift source

在Flume和Kafka发布之前，Thrift和Scribe曾是（现在仍然是）新基础架构中日志采集的主流。使用Thrift source可以让Flume与已有的Thrift/Scribe流水线集成。将type设置为thrift之后，你还必须设置地址绑定和端口。与Avro source类似，将地址绑定设置为0.0.0.0时要小心：

```
agent_1.source.source-1.type=thrift
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

4.3.5.3 Netcat source

Netcat source比Avro和Thrift source都要简单。有的人可能对netcat不太熟悉。在多数类UNIX操作系统中都有该工具。它能用与cat命令读写文件描述符几乎相同的方式读写原始的TCP socket。

与Thrift和Avro source类似，将type设置为netcat之后，也要将Netcat source绑定到特定地址和端口：

```
agent_1.source.source-1.type=netcat
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

Netcat source逐行读取以换行符间隔的文本行，文本行的默认最大长度是512字节（这里不是以字符为单位的原因是字符的长度是可变的）。要将这个长度限制放宽，就要修改max-line-length属性：

```
agent_1.source.source-1.max-line-length=1024
```

为了提供对backpressure的支持，Netcat source还用OK响应来正面确认已收到的每个事件。如果需要，你可以通过将ack-every-event属性设置为false来关闭该acknowledgement：

```
agent_1.source.source-1.ack-every-event=false
```

4.3.5.4 Syslog source

Syslog是系统日志记录的标准，最初是在20世纪80年代作为Sendmail软件包的一部分开发出来的，后来它就成为标准。最初它是由RFC 3164“The BSD Syslog Protocol”规定的，现在由RFC 5424“The Syslog Protocol”规定。

Flume通过Syslog source的三个变种支持RFC 3164和大部分RFC 5424。第一个是称为syslogtcp的单端口TCP source。与其他基于TCP的source类似，它使用常见的绑定地址和端口：

```
agent_1.source.source-1.type=syslogtcp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=5140
```

事件的最大大小是由eventSize属性控制的，默认是2500字节：

```
agent_1.source.source-1.eventSize=2500
```

Flume也支持TCP Syslog source的多端口实现，该实现更现代一些。为此，应该将type改为multiport_syslogtcp，并在配置source时使用ports属性，而不是port属性：

```
agent_1.source.source-1.type=multiport_syslogtcp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=10001 10002 10003
```

Syslog source会以高效方式监控所有端口的syslog事件。除了单端口版本支持的eventSize属性，多端口版本还具备控制字符集的能力，这些字符集用来逐个端口解释事件。默认的字符集通常是UTF-8，它是由

charset.default属性控制的。要使用其他字符集，可以通过charset.port.＜port＞来设置。下面这个例子将默认字符集设为UTF-16，端口10003使用UTF-8：

```
agent_1.source.source-1.charset.default=UTF-16
agent_1.source.source-1.charset.port.10003=UTF-8
```

Syslog source是用Apache Mina实现的，该软件库有一些调优参数可以进行设置。一般情况下不需要修改这些参数，但在某些虚拟环境中可能有必要修改。第一组参数是批处理大小和读缓冲区大小，默认值分别为100个事件和1024字节：

```
agent_1.source.source-1.batchSize=100
agent_1.source.source-1.readBufferSize=1024
```

也可以通过配置来控制处理器的数量。Syslog source为每个处理器派生两个读线程，并尝试自动检测可用的处理器数量。在虚拟环境中，这很可能引发问题，因为字面上的处理器数量并不一定等于实际可用的处理器数量。如果不用自动检测，那就用numProcessors属性来设置处理器数量：

```
agent_1.source.source-1.numProcessors=4
```

最后，Syslog source也有基于UDP的source。对于这样的source，将type设置为syslogudp后，只需要设置地址绑定和端口即可：

```
agent_1.source.source-1.type=syslogudp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=5140
```

该source不再需要其他任何选项，这是因为它使用的是UDP，因此通

常只适合于不需要可靠事件交付的场合。

4.3.5.5 HTTP source

HTTP source使得Flume可以从发出HTTP请求的服务那里获得事件。对于许多可能的事件source来说，这让它们还可以向事件加入丰富的元数据头，而不必实现定制source。与其他所有基于TCP的source类似，该source需要设置绑定和端口，并将type属性设置为http:

```
agent_1.source.source-1.type=http
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=8000
```

通过POST命令，事件被传输给HTTP source，然后返回200或503响应码。返回503响应码说明channel已经满了或不可用。应用接收到该事件后，应该通过其他机制重新尝试发送事件。

事件到达HTTP source后，必须被转换为Flume事件。这是用可插拔（pluggable）的Handler类完成的，为此Flume提供了JSONHandler和BlobHandler类。Handler类默认情况下是JSONHandler，它由handler属性控制。该属性的值应为实现了HTTPSourceHandler接口的类的完全限定类名：

```
agent_1.source.source-1.handler=org.apache.flume.source.http.JSONHandler
```

JSONHandler处理的是JSON对象数组。这些对象每个都包含一个头对象和一个体对象，分别用来填充Flume事件的头元数据和体字节数组。即使是单个事件，也必须包装为数组来处理，如下例所示：

```
[{
  "headers":{
    "timestamp": "34746432",
    "source": "somehost.com"
  },
  "body":"the body goes here."
}]
```

注意，每个头元素的值必须是字符串，即使实际值是数字也应以字符串处理。Flume事件对象的头部元数据只支持字符串。

BlobHandler要简单地多，它只处理较大的二进制对象。该handler不解释事件。

警告　事件是缓存在内存中的，因此特别大的元素可能会将分配给Flume进程的Java堆（heap）填满。这可能导致Flume崩溃，并且根据channel的情况的不同，有可能引起数据丢失。因此，对发送到BlobHandler的对象大小加以限制，这一点很重要。

4.3.5.6　JMS source

Java消息服务（Java Message Service，JMS）source使得Flume可以与基于Java的队列系统集成。当整个软件栈都基于Java时，这些系统就是传送数据的主流机制。JMS source主要是在ActiveMQ上测试的，不过它应该能支持任何JMS提供商。

连接JMS提供商需要对JMS有一定的了解。它至少需要初始化上下文工厂名、连接工厂和JMS提供商的URL。另外还需要目的名称和目的类型，它们可以是队列，也可以是主题。例如，要连接本地ActiveMQ服务器的名为DATA的队列，可以使用下列配置：

```
agent_1.source.source-1.type=jms
agent_1.source.source-1.initialContextFactory=
  org.apache.activemq.jndi.ActiveMQInitialContextFactory

agent_1.source.source-1.connectionFactory= GenericConnectionFactory
agent_1.source.source-1.providerURL= tcp://localhost:61616
agent_1.source.source-1.destinationName= DATA
agent_1.source.source-1.destinationType= QUEUE
```

JMS source支持转换器（converter）的概念，它可以将来自JMS队列的输入转换为Flume事件。尽管可以实现定制转换器，但默认转换器就能够将包含对象、字节或文本的消息转换为本地Flume事件。JMS消息的属性则被转换为Flume事件的头部。

4.3.5.7 Exec source

Exec source允许将任何命令用作流的输入。它常用来仿真Tail Source，后者存在于1.0版本之前的Flume中。正常情况下，要使用该source，只要将type设置为exec，并设置command属性就足够了。例如，下列exec source为agent_1运行tail命令来输出日志文件：

```
agent_1.source.source-1.type= exec
agent_1.source.source-1.command= tail -F /var/log/logfile.log
```

-F开关告诉tail命令要负责日志轮换（log rotation），应该用该选项替代更常见的-f选项。还有许多可选的参数，可以在该source中设置。对于需要shell的命令，可以用shell属性来指定所用的shell。这样我们就可以使用通配符和管道等shell特性：

```
agent_1.source.source-1.shell= /bin/sh -c
```

命令退出后，该source可以重新启动。默认情况下不是这样的，可以

通过将restart属性设置为true来打开该选项：

```
agent_1.source.source-1.restart= true
```

重启命令后，restartThrottle属性负责控制命令重启的速度（以毫秒计）：

```
agent_1.source.source-1.restartThrottle= 1000
```

正常情况下，Exec source只从标准输入中读取事件。也可以通过设置logStdErr属性使其从标准错误流（standard error stream）中读取事件：

```
agent_1.source.source-1.logStdErr= true
```

为了提高性能，Exec source批量读取行，然后将其发送到为其指定的channel。每一批包含的行数默认情况下是20，这是由batchSize属性控制的：

```
agent_1.source.source-1.batchSize= 20
```

单向的source不支持“BACKPRESSURE”

如果可能的话，尽量避免在生产环境中使用Exec这样的source。尽管Exec这样的source非常具有吸引力，但它们会使得Flume框架给出的可靠性保证变得毫无意义。该框架没有与事件产生者进行通信的机制。它不能报告channel已经填满或无法使用的情况，最终结果就会是事件的产生者继续盲目地向source写入事件，导致数据可能丢失。幸运的是，最常见的用例是仿真Tail source（该source因为某种原因被删除了），这时用Spool Directory source来实现通常更好。

4.3.5.8 Spool Directory source

对于不使用数据移动系统的服务，最常见的数据记录方法是将数据日志记录到文件系统中。Flume的早期版本曾尝试通过实现Tail source来利用这一方法，从而让Flume可以在日志写入的时候监视日志并将其移往Flume。可惜的是，如果我们的目的是提供一个可靠的数据移动流水线，这种方法就会有很多问题。为此，Spool Directory source尝试通过支持最常见的轮换日志来克服Tail source的局限性。

要使用Spool Directory source，应用程序必须设置日志轮换。这样，当日志文件前滚时，它会被移动到spool目录。对于多数日志系统来说，这通常很容易做到。Spool Directory source会监控该目录，看是否有新文件，一旦有新文件就将其写到Flume中。要使用Spool Directory source，至少需要定义spoolDir属性，并将type设为spooldir：

```
agent_1.source.source-1.type=spooldir
agent_1.source.source-1.spoolDir=/var/logs/
```

应用程序的常见模式是写到以.log为后缀的文件中，然后将其前滚到以时间戳为后缀的另一个日志文件。通常情况下，活动的日志文件会导致Spool Directory source发生错误，但可以通过设置ignorePattern属性，忽略以.log为后缀的文件，从而忽略这类错误：

```
agent_1.source.source-1.ignorePattern=^.*\.log$
```

当Spool Directory source对文件的处理结束后，它通过向文件名后附加.COMPLETED来标记文件，这是由fileSuffix属性控制的。这使得日志清除工具可以识别出能从文件系统中安全删除的文件。该source也可以通过

将deletePolicy属性设置为immediate，从而在文件处理完成后将其删除：

```
agent_1.source.source-1.fileSuffix=.DONE
agent_1.source.source-1.deletePolicy=immediate
```

默认情况下，该source假设文件中的记录是由换行符间隔的文本。该文本是由Flume自带的LINE反序列化器生成的。基本的Flume软件还自带AVRO和BLOB反序列化器。AVRO根据deserializer.schemaType属性指定的模式，从文件中读取Avro编码的数据。BLOB反序列化器从文件中读取大的二进制对象，通常每个文件只有一个对象。与HTTP BlobHandler类似，这有一定风险，因为会将整个Blob读到内存中。反序列化器的选择是通过deserializer属性来设置的：

```
agent_1.source.source-1.deserializer=LINE
```

如果需要，也可以定制反序列化器。任何实现EventDeserializer.Builder接口的类都能用来替代内置的反序列化器。

4.3.5.9 实现定制source

Flume已经内置了一系列source，因此通常没有必要实现定制的source对象。不过，如果实在有这种需要，是有办法实现的。在Flume中有两类source：轮询source和事件驱动source。轮询source是在Flume自身的线程中运行的，Flume会周期性地查询，将数据移动到相关的channel中。事件驱动source自行维护线程，并异步地将事件放入其channel中。

在这一节中，为了说明连接过程，我们实现了一个连接Flume与Redis的事件驱动source。Redis是一个键-值对存储，我们会在第6章对这种存储

进行深入讨论。除了具有存储功能，Redis还提供临时的发布/订阅接口。我们就是利用该接口来实现source。

定制source总是继承自AbstractSource类，然后实现Pollable或EventDriveSource接口。多数source还实现Configurable接口来处理配置参数。这里，Redis source既实现了Configurable接口又实现了EventDrivenSource接口：

```
package wiley.streaming.flume;

import java.nio.charset.Charset;

import org.apache.flume.Context;
import org.apache.flume.EventDrivenSource;
import org.apache.flume.channel.ChannelProcessor;
import org.apache.flume.conf.Configurable;
import org.apache.flume.event.EventBuilder;
import org.apache.flume.source.AbstractSource;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.JedisPubSub;

public class RedisSource extends AbstractSource
  implements Configurable, EventDrivenSource {
```

Configurable接口有一个配置方法，用来获取Redis的主机名、端口和订阅channel。这里，所有的配置参数都取默认值：

```
  String          redisHost;
  int             redisPort;
  String          redisChannel;
  Jedis           jedis;

  public void configure(Context context) {
    redisHost    = context.getString("redis-host", "localhost");
    redisPort    = context.getInteger("redis-port", 6379);
    redisChannel = context.getString("redis-channel", "flume");
  }
```

EventDrivenSource并不需要真正实现所有方法，它只用来确定source的类型。任何线程甚至处理函数（handler）建立后，都要被分解到start和stop方法中处理。这里，多数工作是在start方法中完成的：

```
@Override
public synchronized void start() {
  super.start();
  processor = getChannelProcessor();
```

首先要获取ChannelProcessor，供后面使用。前面我们曾说过，ChannelProcessor会基于Event来决定使用哪个channel。接下来要完成的订阅线程会使用ChannelProcessor：

```
JedisPool pool = new JedisPool(
  new JedisPoolConfig(),redisHost,redisPort);
jedis = pool.getResource();
new Thread(new Runnable() {
  public void run() {

      jedis.subscribe(pubSub, redisChannel);
    }
  }).start();
}
```

在调用订阅命令时，Redis的Jedis客户端将线程阻塞，后续的所有通信都被发送到JedisPubSub对象。为了避免Flume主线程的阻塞，阻塞操作是在start方法中的独立线程中完成的。JedisPubSub对象定义了多个不同方法，但在这里只有一个方法是必需的：

```
ChannelProcessor processor;
JedisPubSub       pubSub = new JedisPubSub() {

  @Override
  public void onMessage(String arg0, String arg1) {
    processor.processEvent(
      EventBuilder.withBody(arg1, Charset.defaultCharset()));
  }
```

该方法使用EventBuilder类来构造事件，随后该事件就被转发到ChannelProcessor，并放入channel。stop方法从该channel取消订阅，取消对Jedis线程的阻塞，让进程完成处理：

```
@Override
public synchronized void stop() {
  pubSub.unsubscribe();
  super.stop();
}
```

定制source的使用方法与其他source类似。只需要将type属性设为source的完全限定类名，并设置好其他配置参数：

```
agent_1.source.source-1.type=wiley.streaming.flume.RedisSource
agent_1.source.source-1.redis-channel=events
```

4.3.6 Flume sink

Flume sink位于Flume channel的另一端，它们是事件的目的地。Flume自带了大量内置sink，它们多数是向某种持久存储中写入事件的sink。举例来说，其中包括：将数据写到Hadoop的HDFS sink、将事件保存到磁盘的File Roll sink，以及将事件写入Elasticsearch数据存储的Elasticsearch sink。

所有这些sink都是完全成熟的数据流水线的重要组成部分，但它们并不特别适用于流数据处理。正因如此，在这一节中我们只介绍对于流处理更有用的sink，而不会涉及其他sink。

4.3.6.1 Avro sink

Avro sink主要用于在Flume环境中实现多层拓扑。它也是将事件转发到流处理服务进行消费的好方法。Avro sink的type属性应设置为avro，还至少应设置目的主机名和端口：

```
agent_1.sinks.sink-1.type= avro
agent_1.sinks.sink-1.hostname= localhost
agent_1.sinks.sink-1.port= 4141
```

还有许多可选的参数，用来控制sink的行为。可以用相应的属性来控制连接超时时间和重连超时时间，还可以指定重连的间隔时间。这在sink的目的地址是负载均衡连接时可以发挥作用，因为往往只能在连接时进行负载均衡。通过不定期的重连，能够更加平均地分配负载：

```
agent_1.sinks.sink-1.connect-timeout= 20000
agent_1.sinks.sink-1.request-timeout= 10000
agent_1.sinks.sink-1.reset-connection-interval= 600000
```

也可以将compression-type属性设置为deflate来压缩连接，该属性实际上只支持这一个选项：

```
agent_1.sinks.sink-1.compression-type= deflate
agent_1.sinks.sink-1.compression-level= 5
```

4.3.6.2 Thrift sink

Thrift sink与Avro sink基本相同。它也需要指定主机名和端口，并可以用相同的超时行为来控制。不过，它不支持Avro的压缩特性或安全特性。

```
agent_1.sinks.sink-1.type= thrift
agent_1.sinks.sink-1.hostname= localhost
agent_1.sinks.sink-1.port= 4141
```

4.3.6.3 实现定制sink

在与流处理软件通信时，可能有必要实现定制sink。实现定制sink的过程与实现定制source类似。定制的sink类继承自AbstractSink类，可以选择实现Configurable接口。

下面这个例子实现Redis sink来匹配前面提到的Redis source。Redis sink不是向发布-订阅channel订阅事件，而是向channel发布事件。因此，其实现开头的配置与Redis source相同：

```
public class RedisSink extends AbstractSink implements Configurable {
  String          redisHost;
  int             redisPort;
  String          redisChannel;
  Jedis           jedis;

public void configure(Context context) {
  redisHost    = context.getString("redis-host", "localhost");
  redisPort    = context.getInteger("redis-port", 6379);
  redisChannel = context.getString("redis-channel", "flume");
}
```

Redis sink中start和stop方法的实现要简单一些，因为进程与主线程不需要是异步的：

```
@Override
public synchronized void start() {
  super.start();
  pool  = new JedisPool(new JedisPoolConfig(),redisHost,redisPort);
  jedis = pool.getResource();
}

@Override
public synchronized void stop() {
  pool.returnResource(jedis);
  pool.destroy();
  super.stop();
}
```

Process方法从channel中提取事件，将其发送到Redis。简单起见，在下面例子中，我们只传输事件的主体，更精确的sink可能会将输出转换为其他形式，这样就可以保留事件头部的元数据：

```
public Status process() throws EventDeliveryException {
  Channel channel = getChannel();
  Transaction txn = channel.getTransaction();
  Status status = Status.READY;
  try {
    txn.begin();
    Event event = channel.take();
    if(jedis.publish(redisChannel,
      new String(event.getBody(),Charset.defaultCharset())) > 0) {
      txn.commit();
    }
  } catch(Exception e) {
    txn.rollback();
    status = status.BACKOFF;
  } finally {
    txn.close();
  }
  return status;
}
```

4.3.7 sink processor

sink processor是控制channel并行性的机制。实质上，sink processor用来定义sink组。我们可以利用sink组进行负载均衡或故障切换。不过，对于流应用来说，故障切换应用不如负载均衡应用有用。

要对channel激活sink的负载均衡处理，应该将sink的processor.type属性设置为load_balance，并使用processor.sinks属性来定义用于该组的sink：

```
agent_1.sinkgroups= group-1
agent_1.sinkgroups.group-1.sinks= sink-1 sink-2
agent_1.sinkgroups.group-1.processor.type= load_balance
agent_1.sinkgroups.group-1.processor.selector= random
```

上述定义完成之后，channel就被指定给sink组，而不是sink，事件就会随机发送到sink-1或sink-2。负载均衡processor也可以将选择器从random改为round_robin，这样就可以使用轮循机制（round-robin）选择方法。

4.3.8 Flume channel

Flume的channel是source和sink之间的临时存储机制。在基于Flume的生产环境中，有三种不同类型的channel：内存channel、文件channel和JDBC channel。按照这个顺序，这三个channel的安全性越来越高，而性能则不断下降。

4.3.8.1 内存channel

内存channel是所有Flume channel中最快的。同时，它也是最不安全的，因为它将事件存储在内存中，一直到sink将其取走。如果进程由于某种原因被终止，留在内存channel中所有事件都会丢失。

将channel的type属性设置为memory，就可以将其设置为内存channel。此外，还有几个可选的参数。其中最有用的就是capacity和transactionCapacity，它们定义了从channel中消费多少事件才需要加入新事件。这两个参数的取值主要取决于事件的大小和所使用的具体硬件，因此往往需要进行一定的实验来确定给定环境下的最优值：

```
agent_1.channels.channel-1.type= memory
agent_1.channels.channel-1.capacity= 1000
agent_1.channels.channel-1.transactionCapacity= 100
```

4.3.8.2 文件channel

文件channel将事件保存到磁盘，它是最常用的Flume channel。文件channel是为了增加存放于channel中数据的持久性才将数据保存到磁盘的。它使用检查点和数据日志系统，这与Kafka所使用的系统类似。所不同的是，它对事件实行大小约束而不是存放时间约束，在这一点上它与内存channel相同。

多数文件channel可以通过设置channel.type属性来定义，还可能要设置检查点位置属性和数据文件位置属性，如下例所示：

```
agent_1.channels.channel-1.type= file
agent_1.channels.channel-1.checkpointDir= /home/flume/checkpoint
agent_1.channels.channel-1.dataDirs= /home/flume/data
```

类似于Kafka，文件channel可以指定多个数据目录来利用JBOD配置。在实际情况中，使用RAID0配置很可能会让文件channel具有更高的性能，因为这样就能够在存储块层次使写操作并行化，这一点也与Kafka很相似。

4.3.8.3 JDBC channel

尽管Java数据库连接（Java Database Connection，JDBC）channel这个名字看起来跟数据库的访问接口有关，但实际上JDBC channel并不能让FLUME连接任意的数据库。更合适的名字是Derby channel，因为它只允许连接内嵌的Derby数据库。JDBC channel是三个channel中最慢的，但

其可恢复性却是健壮的。要使用它，只需要将channel.type设置为jdbc即可。如果要定义数据库文件的路径，还要设置sysprop.user.home属性：

```
agent_1.channels.channel-1.type= jdbc
agent_1.channels.channel-1.sysprop.user.home= /home/flume/db
```

还有其他属性可以设置，但一般来说这些属性值都不需要修改。

4.3.9 Flume Interceptor

因为Interceptor（拦截器），Flume模糊了数据移动系统和数据处理系统之间的界限。Interceptor模型让Flume不仅可以修改事件的元数据头部信息，还可以根据这些头部信息来过滤事件。

Interceptor绑定在source上，对它的定义和绑定要使用interceptor属性，其方式与channel类似：

```
agent_1.sources.source-1.interceptors= i-1 i-2
```

配置时Interceptor的定义顺序就是它们对事件的应用顺序。由于Interceptor可以将整个事件丢弃，如果过滤Interceptor要使用前面的Interceptor所设置的头部信息，那么在配置时就要留心它们的顺序。

Flume提供了大量有用的Interceptor，可以用来对事件进行包装或在流处理时提供有用的审计信息。本节列出了最常用的几个。

4.3.9.1 时间戳Interceptor

时间戳Interceptor在事件中加入当前的时间戳作为元数据头部，其键

名为timestamp。有一个可选的属性preserveExisting，可以在时间戳已存在时控制是否对其重写：

```
agent_1.sources.source-1.interceptors.i-1.type=timestamp
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

4.3.9.2 主机Interceptor

主机Interceptor向头部元数据中加入Flume agent的主机名或IP地址。默认的头部称作host，你也可以使用hostHeader属性修改该头部。它还支持preserveExisting属性：

```
agent_1.sources.source-1.interceptors.i-1.type=host
agent_1.sources.source-1.interceptors.i-1.hostHeader=agent
agent_1.sources.source-1.interceptors.i-1.useIP=false
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

该Interceptor对于在流处理时保持审计跟踪（audit trail）很有用。

4.3.9.3 静态Interceptor

静态Interceptor用来向所有事件添加静态头部。它有key和value两个属性，都用来设置头部：

```
agent_1.sources.source-1.interceptors.i-1.type=static
agent_1.sources.source-1.interceptors.i-1.key=agent
agent_1.sources.source-1.interceptors.i-1.value=ninetynine
```

该Interceptor主要用于识别采用不同处理路径的事件。让agent向头部中加入路径信息，随后我们就可以在整个系统中对事件进行跟踪。

4.3.9.4 UUID Interceptor

这种Interceptor向它处理的每个事件加入一个全局唯一的标识符。默

认情况下它将头部名设为id，不过你当然可以改成其他名字：

```
agent_1.sources.source-1.interceptors.i-1.type=uuid
agent_1.sources.source-1.interceptors.i-1.headerName=id
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

正如我们在本章开头所讲到的，这些系统多数实现的是至少一次的交付语义，这有时会导致出现重复事件。根据应用的具体情况，有时有必要对重复事件进行检测和去重。将这个Interceptor放在Flume拓扑的开头，就可以唯一地识别事件。

4.3.9.5　通过正则表达式进行过滤的Interceptor

正则表达式过滤Interceptor将事件的主体看作字符串，对其使用正则表达式。如果正则表达式匹配成功，就根据excludeEvents属性保留或丢弃该事件。如果excludeEvents为true，就丢弃匹配正则表达式的事件。若为false，就只保留匹配正则表达式的事件。通过定义regex属性来定义正则表达式：

```
agent_1.sources.source-1.interceptors.i-1.type=regex_filter
agent_1.sources.source-1.interceptors.i-1.regex=.*
agent_1.sources.source-1.interceptors.i-1.excludeEvents=false
```

该Interceptor可以用来为不同类型的事件设置不同路径。当事件处于不同路径时，不需要在处理代码加入任何if-else语句，就可以很容易地处理事件。

4.3.10　集成定制Flume组件

上一节我们提到过，Flume的许多组件都可以替换为定制实现。这些

定制组件的开发通常与Flume源代码无关。有两种集成方法可供使用。

4.3.10.1　插件集成

最值得推荐的集成方法是通过Flume的插件功能进行集成。要使用这种方法，需要在Flume的主目录中创建一个名为plugin.d的目录。要包含的每个插件都要放到plugin.d下的一个子目录中。例如，要包含本书的例子，plugin.d的结构应该如下所示：

```
plugin.d/
plugin.d/wiley-plugin/
plugin.d/wiley-plugin/lib/wiley-chapter-4.jar
```

如果还需要更进一步的依赖，可以将它们放到插件的libext子目录，或者直接构建到所包含的jar包中。本地库放在native子目录中。当Flume agent启动时，它会扫描插件目录，将它在其中找到的所有jar文件都加入类路径（classpath）。

4.3.10.2　类路径集成

如果由于某种原因不能使用插件集成，Flume还可以使用更标准的基于类路径的集成方法。要使用这种方法，就应该在启动Flume agent时，用FLUME_CLASSPATH环境变量将库加入Flume agent，以供将来使用。

4.3.11　运行Flume agent

Flume agent的启动是通过flume-ng脚本完成的，该脚本位于Flume安装目录。启动脚本需要一个名字、一个配置目录和一个配置属性文件（有

的配置选项可能会引用配置目录中的其他文件）：

```
$ ./bin/flume-ng agent --conf conf \
>   -f conf/agent.properties --name agent-1
```

agent命令告诉Flume它正在启动一个agent。--conf指定配置目录，-f指定包含agent定义的属性文件。-name告诉Flume启动名为agent-1的agent。agent启动后，就根据自己的配置开始处理事件。

4.4 总结

本章讨论了在分布式环境下处理数据流程的两种系统：Kafka和Flume。它们都是高性能的系统，都具有支持实时流所需要的足够的可扩展性。Kafka直接面向从头开始构建应用的用户，能够给用户自由来直接集成健壮的数据移动系统。Flume也可以用于新的应用，但由于其设计的原因，它最适合于将已有的应用集成到一个处理环境的情形。

说到这里，似乎这两个系统有点互斥：Kafka适合新应用，而Flume适合“遗留”应用。其实并不是这样。这两个系统是互补的，它们各自面向不同的使用模式。在复杂环境下，Kafka可能用于系统中的批量数据移动，负责批量日志记录和对外部数据源的数据采集。它也可能用来直接向第5章的流处理系统提供输入。不过，Flume更适合于将数据流输入到ElasticSearch这样的持久数据存储。如果用Flume，这些组件是现成的，可以直接使用，而如果用Kafka，就要从头开发组件。

现在数据已经在系统中流动，是时候对其进行处理和存储了。后面的过程原本是相当复杂的。还好，第5章介绍的系统能够极大地简化这些机制。

第5章　流数据的处理

数据流过数据采集系统之后，就必须得到处理。Kafka和Flume的最初用例都将Hadoop作为处理系统。Hadoop是批处理系统。尽管它擅长批处理，但其处理速度却难以保证延迟低于5秒。

批处理速度的限制主要源于启动和关闭的开销。当Hadoop作业启动时，首先从输入源（通常是Hadoop分布式文件系统（Hadoop Distributed File System，HDFS），但也可能是其他数据源）获取一组输入分片（input split），然后由Job Tracker将输入分片打包进各个mapper任务中，这些任务有可能还会在worker节点启动新的虚拟机实例。接下来进入shuffle、sort和reduce阶段。

尽管这些步骤每个都很小，但加起来却很耗时。根据集群性质的不同，典型的任务启动需要10~30秒的“墙上时间”。Hadoop 2的启动总时间实际上要更长，因为它要启动一个应用程序管理器来管理作业。对于将要运行30分钟或1小时的批处理作业来说，这样的启动时间对于性能调优微不足道，完全可以忽略。但是如果是每5分钟运行1次的作业，30秒的启动时间就意味着10%的性能损失。

本章的主题是实时处理框架，它们通过使用长时间运行的进程和非常小的批处理包（可能跟单个记录一样小，但通常要大一些）来规避启动和关闭延迟。一个作业可能运行几小时、几天甚至几周才重启，这样就将启动开销分摊为0。这种方法的缺点是会引入一些管理开销，这是为了保证作业能在很长时间内正常运行，以及处理作业不同部分之间的通信，以便

利用多台机器并行处理。

我们首先讨论影响流数据框架开发的几个问题。流数据分析问题不是一个新话题，它比“大数据”的兴起还要早。

然后深入讨论可以用来实现流系统的两个完整的处理框架：Storm和Samza。无论利用这两个框架中的哪一个，你都可以搭建服务器基础架构，使其支持分布式处理，并定义结构良好的API来描述这些服务器是怎样执行分布式计算的。

5.1　分布式流数据处理

对在特定框架内实现流处理应用的细节进行深入讨论之前，花点时间来理解分布式流处理的工作原理是值得的。对于某些问题来说，流处理是完美的解决方案，但有一些需求会影响流处理性能，甚至会导致流处理系统的处理速度不如批处理系统。当处理过程中各单元之间需要进行协调时，这种情况特别容易出现。

本节介绍流处理系统的几个特性。流处理本质上是一种特殊形式的并行计算，它被设计用于数据只能处理一次的情况。但是，与单体式的超级计算机时代不同，这些并行计算环境多数实现在一个不可靠的网络层之上，这样的网络层会引入高得多的错误率。

5.1.1　协调

所有流框架的核心都是某种协调服务器。这些协调服务器存储用于拓扑处理的相关信息，包括哪个组件应该获取什么数据，以及它们占用哪个物理地址。

协调服务器还要维护一个分布式锁服务器，来处理一些分区任务。具体来说，刚开始将数据加载到分布式处理框架时需要用到锁服务器。数据的读取速度往往远大于处理速度，因此，即使是数据移动系统的内部分区也可能需要使用多个进程来从给定分区中抽取数据。这就需要客户端间进行协调。

本章讨论的两个框架都使用ZooKeeper来处理各自的协调任务。根据负载的不同，可以对用来管理数据移动框架的ZooKeeper集群进行重用，这样做没有多少风险。

5.1.2　分区和融合

流处理系统的核心元素是分散-收集（scatter-gather）机制的某种实现。

输入流被划分到许多相同的处理流水线中。根据流水线中每个步骤性能特性的不同，计算结果可能会被进一步细分，以维持性能不变。

例如，如果步骤2需要的时间是步骤1的两倍，那就有必要将流水线中指定给步骤2的工作单元数量至少增加为步骤1的两倍。通常需要将单元数量增加为2倍以上，这是因为工作单元之间的通信会引入处理开销。

流处理应用的各种组件之间的数据交换通常是点对点形式。一个组件知道它必须生成数据供别的组件消费，多数系统会建立处理单元间的数据传送机制，并在计算的每个阶段控制并行度。

5.1.3　事务

事务处理是实时系统的一个重要问题。事务的维护会给系统带来巨大开销。如果没有事务也可以运行，那就不要用事务。如果这无法实现，一些框架在相应的数据移动系统的支持下可以提供某种层次的事务支持。

实时框架中事务处理的基本机制是回滚和重试。框架首先生成输入流的一个检查点（checkpoint）。这在队列系统或数据移动系统中是很简单的，因为这些系统中的每个元素都有一个唯一的标识符，这些标识符在相关的时间范围内是单调递增的。在Kafka中的实现方法是，记录每个分区在主题（topic）中的偏移量，这里的每个分区都是实时处理系统的输入。

5.2 用Storm处理数据

Storm是营销分析公司BackType开发的流处理框架，该公司曾为Twitter分析数据。Twitter于2011年收购了BackType，采用BackType的核心技术Storm，并最终将其开源。Storm目前的版本是0.9.0，本节我们关注这个版本，因为它有一些新特性能够简化Storm的用例。

Storm主要聚焦于容错的分布式流计算的开发和部署。为此，它提供了一个框架来支撑应用，还提供了两个模型来构建应用。

第一个模型是“经典”Storm，它将应用构造为名为拓扑（topology）的有向无环图（Directed Acyclic Graph，DAG）。该图的顶部从外部（例如Kafka）获得输入，这些数据源称为spout。然后由这些spout将数据传递到称为bolt的计算单元中。

第二个用于构建应用的模型称为Trident，是对拓扑的高层次抽象。该模型更聚焦于聚集和持久化这些常见操作。它提供了针对这类计算的原语，并根据按照处理顺序来排列这些原语。然后Trident就将操作合并和分割到相应的blot集合中，来计算该拓扑。

本章介绍了这两个计算模型。一般来说，我们推荐使用第二个模型（使用Trident的特定领域语言），因为它标准化了许多常见的流处理任务。不过有的时候，不用这种常见模型，转而使用底层的拓扑框架反倒更有优势。

5.2.1　Storm集群的组件

如果要在生产中部署Storm拓扑，那就必须构建一个集群。该集群由三个服务器守护进程组成，分别是ZooKeeper、nimbus和supervisor，这三个进程都必须正常运行。在这一节中，我们将用这三个组件来构建能支持生产环境的工作负载的Storm集群，如图5-1所示。

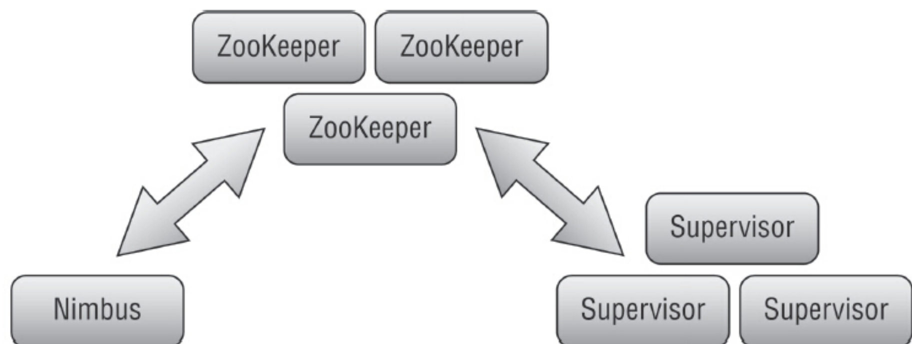


图　5-1

5.2.1.1　ZooKeeper

分布式Storm集群依靠ZooKeeper进行协调。由于节点间的通信是点对点的，所以ZooKeeper集群的工作负载并不会很重，这是因为它只用于管理每个节点的元数据。如果要使用同样依靠ZooKeeper的数据移动系统，例如Kafka，只要Storm集群有足够的容量，使用同一个集群也是可以的。

ZooKeeper会跟踪所有运行着的拓扑，以及所有supervisor的状态。

5.2.1.2　nimbus

nimbus是Storm的首要组件。它负责向集群中的worker分配spout或bolt的任务。它还负责在supervisor发生故障时使Storm集群重新平衡。

初看上去，nimbus似乎会像传统Hadoop集群中的JobTracker或NameNode那样，存在单点失效问题。严格来讲，对于Storm并非如此。当nimbus失效时，已有的拓扑会继续运行，不受影响。nimbus并不介入实时处理过程，它只负责向supervisor节点分配处理任务。

当nimbus失效时，集群将无法再管理拓扑。这会启动新的拓扑并对已有的拓扑进行失效时的重新平衡。建议你尽快重启nimbus，但nimbus失效不会立刻导致整个集群宕机。

5.2.1.3　supervisor

Storm中的supervisor与Hadoop的TaskTracker类似。它们是运行于每个worker节点之上的服务器，负责管理本地任务的运行。每个本地任务可能是一个spout或一个bolt，每个supervisor都可能控制许多本地任务的运行。

常规操作下，如果supervisor失效，那么它控制的所有运行中的任务也会全部丢失。得益于Storm的分布式和容错设计，这种情况下，nimbus会在一个supervisor的控制下重启这些任务。

当某个supervisor重返集群（或加入了更多supervisor）时，运行中的拓扑不会自动重新平衡。加入更多节点后，如果要使集群重新平衡，应该在Storm命令行界面上运行rebalance命令。

5.2.2　配置Storm集群

本节我们演示怎样配置Storm集群进行分布式模式的处理。在分布式硬件上，这就是生产环境的处理模式。在单台机器上运行所有服务也是可以的，但这时用下一节介绍的本地集群模式来开发和调试要更容易。

5.2.2.1　准备工作

首先从http://storm-project.net 网站下载Storm的二进制安装包。撰写本书时该安装包的最新版本是0.9.0.1，我们推荐安装这个版本。

如果由于某种原因你必须使用旧版本的Storm，那就安装ZeroMQ（也写作0MQ）。由于支持Netty传输机制，0.9.0版本无须安装ZeroMQ。但是Storm的早期版本需要这个软件包才能让节点可以互相通信。

在Linux系统中，可以通过软件包管理系统获得ZeroMQ，但要注意，你应该安装2.1.7版本，这是已知支持Storm的唯一版本。Mac OS用户可以从Homebrew或MacPorts等软件包库中获取0MQ。Windows用户则会遇到很多问题。相比在Windows上运行旧版本Storm，直接升级到0.9.0版本更容易。

5.2.2.2　配置Storm

Storm将基本的配置信息放在conf/storm.yaml文件中。该文件不包含拓扑的相关信息，但会记录一些重要特性，包括ZooKeeper服务器的位置、nimbus服务器的位置，以及传输机制。

假设目前运行了一个与第3章中配置相同的ZooKeeper集群。像所有的ZooKeeper客户端一样，Storm需要知道至少一个ZooKeeper服务器的名称。如果有多个服务器，在一个服务器临时发生故障时就可以进行故障切换。这些服务器是用YAML列表在storm.zookeeper.servers配置参数中定义的：

```
storm.zookeeper.servers:
  - "zookeeper-node1.mydomain.net"
  - "zookeeper-node2.mydomain.net"
```

如果是在单台机器上运行一个分布式服务器，那在这里只需要用localhost：

```
storm.zookeeper.servers:
  - "localhost"
```

为了能够提交拓扑，Storm还需要知道nimbus服务器的位置：

```
nimbus.host: "storm-nimbus.mydomain.net"
```

如果所有数据处理都是本地运行的，这里就应该设置为localhost：

```
nimbus.host: "localhost"
```

如果使用Storm 0.9.0，那就不要使用ZeroMQ传输机制，而应该用Netty传输机制。后者是由Yahoo！开源的，它能提供更好的性能，启动也更容易，应该作为所有集群的默认传输机制。不过，它并不是Storm集群的默认机制，因此需要在配置文件中加入下列语句来配置：

```
storm.messaging.transport: "backtype.storm.messaging.netty.Context"
storm.messaging.netty.server_worker_threads: 1
storm.messaging.netty.client_worker_threads: 1
storm.messaging.netty.buffer_size: 5242880
storm.messaging.netty.max_retries: 100
storm.messaging.netty.max_wait_ms: 1000
storm.messaging.netty.min_wait_ms: 100
```

实际上，上述修改并不仅仅加入了对Netty的支持。从技术上来说，Storm现在支持加入其他类型的传输机制。

5.2.3　分布式集群

Storm的默认模式是以分布式服务器集群的形式运行。其中一台机器运行nimbus，通常还要运行Storm用户界面（UI）。其他机器运行supervisor，并根据容量扩展的需要来添加新机器。本节介绍启动这类集群需要的步骤，尽管所有的“机器”实际上都运行在同一个开发环境中。

5.2.3.1　启动服务器

Storm包含一个可以运行的开发版的ZooKeeper安装包，如果这是一个开发环境中的机器，应该在启动其他服务器之前启动它：

```
storm-0.9.0.1$ ./bin/storm dev-zookeeper
```

当它退出连接而掉线时，在它自己的控制台中使其保持运行很有用。

现在就可以启动所有服务器了。在要运行nimbus的服务器上使用bin/storm nimbus命令来启动nimbus服务器。此外，还要启动Web界面，该界面用来管理拓扑。

```
storm-0.9.0.1$ ./bin/storm nimbus &
storm-0.9.0.1$ ./bin/storm ui &
```

当然，在生产环境中，这些命令应该放到某种启动脚本中。

相类似，应该在所有的worker机器上启动Storm supervisor：

```
storm-0.9.0.1$ ./bin/storm supervisor &
```

如果一切顺利，Storm UI就会显示出supervisor的正确数量。例如，运行在开发环境中的机器上的集群应该如图5-2所示：

5.2.3.2　控制拓扑

向Storm集群提交拓扑与提交Hadoop作业类似。其命令也是相似的：只需要将hadoop jar改为storm jar。

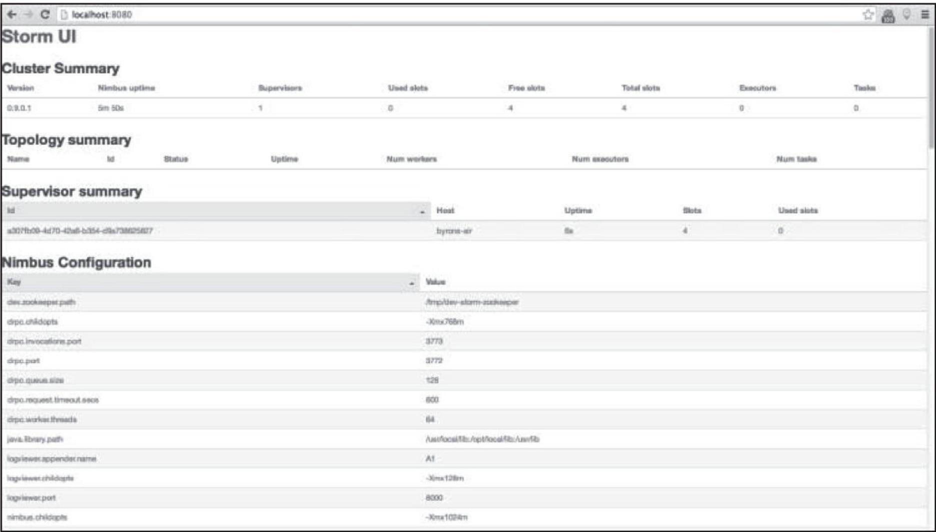


图　5-2

拓扑本身实际上是在其主类（main class）中使用StormSubmitter类提交的：

```
Config conf = new Config();
StormSubmitter.submitTopology("my-topology", conf, topology);
```

可以原封不动地使用Config对象，也可以将其重写来设置特定拓扑的配置参数。在恰当的配置上下文中，可以使用storm jar命令运行实现该调用的主类：

```
$ ./bin/storm jar \
> streaming-chapter-5-1-shaded.jar \
> wiley.streaming.storm.ExampleTopology
246 [main] INFO backtype.storm.StormSubmitter
  - Jar not uploaded to master yet. Submitting jar...
346 [main] INFO backtype.storm.StormSubmitter - Uploading topology
  jar streaming-chapter-5-1-shaded.jar to
  assigned location: storm-local/nimbus/inbox/stormjar-18e1fded
  -074d-4e51-a2b1-5c98b2e133a6.jar
1337 [main] INFO backtype.storm.StormSubmitter - Successfully uploaded
  topology jar to assigned location:
  storm-local/nimbus/inbox/stormjar-18e1fded-074d
  -4e51-a2b1-5c98b2e133a6.jar
1338 [main] INFO backtype.storm.StormSubmitter - Submitting topology
  my-topology in distributed mode with conf {}
```

提交拓扑之后，可以用Storm命令行客户端来终止拓扑。可以用kill命令指定拓扑标识符来完全终止拓扑。可以用deactivate和activate命令在不终止拓扑的情况下分别暂停和重启拓扑。也可以在Storm的Web界面上从UI中选择相应的拓扑将其暂停和重启。

5.2.3.3 启动拓扑工程

要启动拓扑工程的开发工作，最简单的方式是使用Maven来包含所有的必备依赖。下面所示的是启动一个基于Java的Storm拓扑所需要的最小

的pom.xml。它包含了最小的依赖，其中包括到clojars.org资源库的链接，这个资源库中有Storm的maven包。它还包括shade插件，用来构建所谓的“fat jars”，从而将Storm自身没有包含的其他依赖添加进来。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>group</groupId>
  <artifactId>storm-project</artifactId>
  <version>1</version>
  <repositories>
  <repository>
    <id>clojars.org</id>
    <url>http://clojars.org/repo</url>
  </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>storm</groupId>
      <artifactId>storm</artifactId>
      <version>0.9.0.1</version>
      <scope>test</scope>
    </dependency>
</dependencies>
  <build>
    <plugins>
      <plugin> <!-- Shade Plugin for building Fat Jars -->
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <shadedArtifactAttached>true</shadedArtifactAttached>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals><goal>shade</goal></goals>
         </execution>
       </executions>
     </plugin>
   </plugins>
 </build>

</project>
```

以上就是用Java启动Storm拓扑的开发所需要的全部工作。

5.2.4 本地集群

在开发和测试Storm拓扑时，没有必要使用完整的集群。完整的集群对跟踪软件包问题倒是很有用。

为了便于测试，Storm提供了一个称为本地集群（LocalCluster）的特殊类型集群，这样就可以在单个Java虚拟机（JVM）实例上运行整个拓扑。这个集群是“完整”的，因为它在一个supervisor内部运行，并且运行着一个内置的ZooKeeper实例。这与单元测试环境形成鲜明对照，后者的服务器组件是“模拟”出来的。

要在测试框架中使用本地集群是非常简单的，只需要配置一个LocalCluster，然后像往常一样将拓扑提交给它即可：

```
Config conf = new Config();
conf.setDebug(true);
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("example", conf, builder.createTopology());
Thread.sleep(60000);
```

这里需要使用sleep命令，这是因为集群拓扑实际上是在后台线程上运行的。如果不用该命令，主方法就会结束，进程将立刻关闭，不会实际地运行该拓扑。向主类中加入延迟就可以让拓扑运行一小段时间。

5.2.5 Storm拓扑

拓扑是Storm的计算组织机制。它是用有向无环图（DAG）来表示的。

众所周知，图是由顶点（节点）以边相互连接而成的结构。在这里，边是有向的，这意味着数据只能沿着边单向流动。

这里的图还是无环的，这说明边的连接不会形成回路。否则，数据会在拓扑中无限流动。图5-3给出了一个简单的拓扑，包括一层输入spout和两层bolt。为了保持性能，Storm可以根据需要增加blot的并行性。

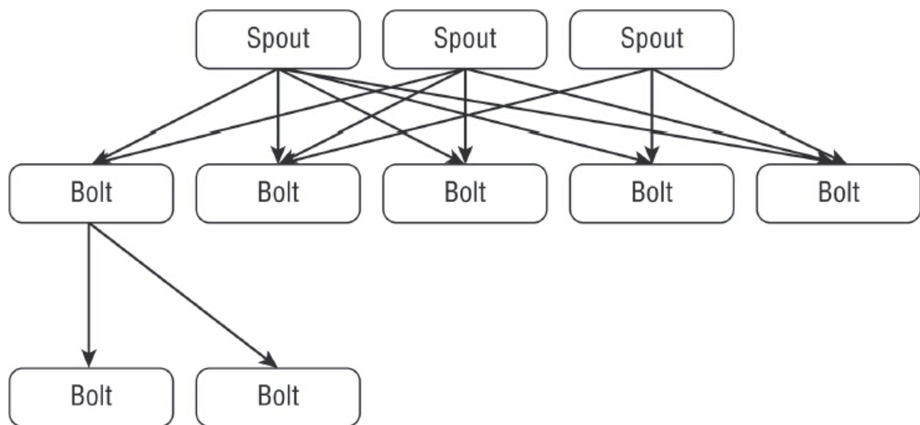


图　5-3

要在Storm中创建拓扑，要用TopologyBuilder类，调用其createTopology方法来生成拓扑：

```
public static void main(String[] args) {
  TopologyBuilder builder = new TopologyBuilder();
  defineTopology(builder,args);
  StormTopology topology  = builder.createTopology();
```

defineTopology方法用来构造图本身。拓扑中的每个顶点由一个唯一的名字和一个spout或bolt的定义组成。spout是Storm的数据输入机制，添加spout需要使用setSpout方法：

```
public static void defineTopology(TopologyBuilder builder
   ,String[] args) {
    builder.setSpout("input", new BasicSpout());
```

这里生成了一个名为“input”的spout，并返回一个SpoutDeclarer，后者用来进一步配置spout。多数拓扑只包含一个IRichSpout类型，但是Storm往往生成多个spout任务来提高输入过程的并行性和持久性。任务的最大数量以及其他相关的变量都是用返回的SpoutDeclarer来设置的。

bolt是Storm的基本计算单元。与spout类似，bolt必须有唯一的名字。通过setBolt方法可以添加bolt。例如，下列代码向拓扑中加入一个名为“processing”的新bolt：

```
builder.setBolt("processing", new BasicBolt())
       .shuffleGrouping("input");
```

SetBolt方法返回BoltDeclarer对象，这个对象用来将blot与拓扑的其他部分联系起来。bolt是用其中一个分组方法（grouping method）来定义一个输入顶点，这样就与拓扑相连。这时就要用到先前定义的“输入”spout顶点。

Storm为每个已定义好的bolt生成多个任务来提高并行性和持久性。这可能影响到聚集等计算，因此要用分组方法来定义数据怎样发送到各个任务。这与Map-Reduce实现中分区（partition）函数的用法类似，但要略复杂。Storm提供了许多分组方法来组织拓扑中的数据流程。

5.2.5.1 随机分组

先前的例子中使用shuffleGrouping方法在bolt任务中分发数据。该方法在bolt任务中随机分发称为元组（Tuple）的数据元素，从而让每个任务

得到几乎等量的数据。

5.2.5.2　字段分组

另一个常见的分组方法是fieldGrouping方法。该分组使用一个或多个由输入顶点定义的已命名的元组元素，来确定由哪个任务接收特定的数据元素。它实质上等价于SQL的GROUP BY子句，经常在实现聚集blot时使用。例如，要根据key1和key2来从“输入”中对数据分组，应按照如下方式加入bolt：

```
builder.setBolt("processing", new BasicBolt())
  .fieldsGrouping("input", new Fields("key1","key2"));
```

5.2.5.3　全部分组和全局分组

allGrouping和globalGrouping方法这两者恰恰相反。前者确保对于给定bolt，事件数据流中的所有元组都传输到所有运行的任务中。该方法在少数情况下是有用的，但它让事件的数量暴增（事件翻倍数为运行任务的数量），因此要小心使用。

globalGrouping方法与之相反。它确保流中的所有元组都送到具有最小标识符的bolt。所有的bolt任务都会收到一个标识符。该方法确保只用其中一个bolt。这个方法也要谨慎使用，因为它实际上使Storm丧失了并行性。

5.2.5.4　直接分组

directGrouping方法是一种特殊形式的分组，它让输出bolt或spout决定哪个bolt任务接收特定的Tuple。这需要在生成bolt时将流声明为直接

的。生产者bolt也必须使用emitDirect方法，而不是emit方法。

emitDirect方法还有另外一个参数，用来确定目的bolt。可以在实现bolt时从Topology-Context获取可用的标识符列表。具体细节我们会在5.2.6节中介绍。

5.2.5.5　自定义分组

当其他分组方法都不合用时，还可以实现自定义的分组方法。为此，要创建一个实现CustomStreamGrouping接口的类。

该接口包含两个方法。第一个方法prepare是在拓扑实例化时调用的，它让分组方法收集执行任务所需的所有元数据。具体来说，该方法接收targetTasks变量，该变量是订阅到要分组的流的bolt列表。

第二个方法是chooseTasks，它是类中真正实现分组的方法。每个Tuple都会调用该方法。这个方法会返回一个应该接收Tuple的bolt任务列表。

一个轮循机制的自定义流分组方法

这个例子实现了在bolt任务中分发数据的轮循（round-robin）机制。该机制并不比shuffleGrouping更高效，我们主要用它进行技术演示。

首先，用序列化的版本号来定义类。Storm在配置拓扑时会频繁使用Java序列化，因此你要仔细注意哪些应该序列化，哪些不应序列化，这很重要。该类的三个字段都是在prepare语句中设置的，因此应将它们定义为transient（临时的），这样它们就不会被包含在序列化中：

```java
public class RoundRobinGrouping implements CustomStreamGrouping {

  /**
    *
    */
   private static final long serialVersionUID = 1L;


   transient List<Integer> targetTasks;
   transient int           nextTask;
   transient int           numTasks;
```

接下来，prepare语句将每个临时变量初始化。更复杂的分组方法可能会检查拓扑自身，从而为任务赋予权重和其他选项，但对这个简单的轮循方法来说，只需要保存任务列表：

```java
public void prepare(WorkerTopologyContext context,
          GlobalStreamId stream,
          List<Integer> targetTasks) {
    this.targetTasks = targetTasks;
    this.numTasks    = targetTasks.size();
    this.nextTask    = 0;

  }
```

最后，实现chooseTasks方法，在各任务中循环：

```java
public List<Integer> chooseTasks(int taskId, List<Object> values) {
     LinkedList<Integer> out = new LinkedList<Integer>();
     out.add(targetTasks.get(nextTask));
     nextTask = (nextTask + 1) % numTasks;
     return out;
   }
```

5.2.6 实现bolt

现在数据已经妥善地传递给bolt，是时候理解bolt自身是如何工作的。正如前一节所介绍的，bolt从拓扑中的其他bolt或spout接收数据。它

们是由事件驱动的，因此不能用来提取数据。提取数据由spout负责。

尽管可以通过实现IBasicBolt或IRichBolt接口来实现bolt，但还是建议你通过继承BaseBasicBolt或BaseRichBolt类来实现。对绝大多数Storm样板方法，这些基类都提供了简单实现。

5.2.6.1　Rich bolt

要实现RichBolt，首先应继承RichBaseBolt抽象类，并实现所需要的三个方法。与上一节的自定义分组类相似，第一个就是prepare方法，要将拓扑的相关信息传递给该方法。对bolt来说，还要加入另外一个参数：collector对象。该对象管理bolt和拓扑之间的交互，特别是传输元组和确认元组。

在rich bolt中，collector通常放在实例变量中，已在execute方法中使用。prepare方法也是创建不能序列化的对象的好地方，因为它们不实现Serialization接口。并不是所有变量都需要transient关键字，但它们经常会被无意间初始化或序列化，从而导致发生难以追踪的错误，因此，将prepare方法实例化的所有变量都声明为transient是一个好习惯。下列代码实现了获取输出collector的bolt：

```
public class RichEmptyBolt extends BaseRichBolt {

  private static final long serialVersionUID = 0L;
  private transient OutputCollector collector;
  public void prepare(Map stormConf, TopologyContext context,
                      OutputCollector collector) {
    this.collector = collector;
  }
```

多数bolt会产生一个或多个输出流。流是由Storm序列化的Tuple对象

集合，这些对象被按照拓扑定义传递给其他bolt。这些Tuple对象没有形式化的模式，元组中的所有元素都被视为泛化的Object，但是都有名称。这些名称会在bolt的execute方法和特定的分组类（如fieldGrouping）中用到。这些命名字段的顺序是在bolt的declareOutputFields方法中定义的。默认情况下，bolt有一个数据流，该数据流是调用传入该方法的OutputFieldsDeclarer对象的declare方法得到的。

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("first","second","third"));
}
```

除了默认流，bolt也可以为输出定义其他命名流。使用declareStream方法，就会给流赋一个名字，并定义元组的字段。例如，将默认流分割为两个部分，声明为两个输出流：

```
declarer.declareStream("car", new Fields("first"));
declarer.declareStream("cdr", new Fields("second","third"));
```

定义拓扑时，这些额外的流会被加入source名称后面的分组调用中。例如，用字段分组向cdr流附加一个bolt：

```
builder.setBolt("empty", new RichEmptyBolt());
builder.setBolt("process", new BasicBolt())
  .fieldsGrouping("empty", "cdr", new Fields("second"));
```

输入元组是在bolt内部的execute方法中处理的。在RichBolt实现中有一个Tuple参数。这个类实现Java的List＜Object＞collection接口，也支持对source bolt所声明字段的高层访问。

bolt可以对输入元组做任何处理。例如，要生成前面例子中声明的输出流，假设输入元组的元素名称与输出元组的相同，execute方法的代码如

下所示：

```
public void execute(Tuple input) {
  List<Object> objs = input.select(new Fields("first","second","third"));
  collector.emit(objs);
  collector.emit("car", new Values(objs.get(0)));
  collector.emit("cdr", new Values(objs.get(1),objs.get(2)));
  collector.ack(input);
}
```

这个例子的第一行使用Tuple包装器的一个特性，以事先确定的顺序提取Tuple对象的子集。这个顺序与所声明的输出流的顺序一致，因此它只需要按原样发送数据。emit方法假定输出是实现了List＜Object＞的对象，这个对象在到达下一个bolt后会被转换为Tuple对象。

接下来发送其他两个流：car和cdr。由于这两个流都只包含默认流的一部分，因此需要构造新的输出数组。List对象默认的Java构造器很冗长，为此Storm提供了Values类，用来实现List＜Object＞接口，并提供简便的构造器，从而更易于生成新的Tuple对象。

最后，通过ack（input）对元组进行确认。该方法告诉Storm该元组已经得到处理。当bolt是以事务方式使用时，这一点很重要。如果由于某种原因元组没有得到处理，bolt就可以使用fail（input）替代ack（input），来告知Storm处理未成功。

collector对象还有一个reportError方法，这个方法可以用来报告流中发生的非致命性异常。该方法以一个Throwable对象作为输入参数，异常信息会在Storm UI中显示。例如，处理URL的bolt可能会报告解析错误：

```
public void execute(Tuple input) {
  try {
    URL url = new URL(input.getString(0));
    collector.emit(new Values(url.getAuthority(),
                              url.getHost(),
                              url.getPath()));
  } catch (MalformedURLException e) {
    collector.reportError(e);
    collector.fail(input);
  }
}
```

过滤和拆分bolt

这个例子给出了一个bolt，它接收一个输入流，根据建立在字段上的正则表达式将其拆分为多个不同的流。例子中还展示了如何对输入元组进行内省（introspect），以便使用程序定义输出流。

过滤器是在配置拓扑时定义的。bolt定义了一个名为FilterDefinition的简单的内部类，这个类中包含输出流、要查看的字段名，以及基于字段值的正则表达式。由于实现了Serializable，因此当多个bolt分散在集群中时，该类将被正常地序列化。

Filter函数本身是以“链式调用”风格实现的，这样更易于在拓扑定义中使用：

```
public class FilterBolt extends BaseRichBolt {
  private static final long serialVersionUID = -7739856267277627178L;

  public class FilterDefinition implements Serializable {
    public String  stream;
    public String  fieldName;
    public Pattern regexp;
    public Fields  fields;
    private static final long serialVersionUID = 1L;
  }

  ArrayList<FilterDefinition> filters
  = new ArrayList<FilterDefinition>();

  public FilterBolt filter(String stream,String field,String regexp,
    String... fields) {
    FilterDefinition def = new FilterDefinition();
    def.stream = stream;
    def.fieldName = field;
    def.regexp = Pattern.compile(regexp);
    def.fields = new Fields(fields);
    filters.add(def);
    return this;
  }
```

bolt的prepare方法以collector的输出作为输入，并为过滤器做一些准备工作以备后面使用。这里，要检查唯一的输入流来确定在输出元组中出现的字段。如果这些字段与某个特定的过滤器匹配，该过滤器就与该流绑定。为了提高性能，还获取了过滤器元素的索引：

```
  public class FilterBinding {
    public FilterDefinition filter;
    public int              fieldNdx;
  }


  transient OutputCollector                      collector;
  transient Map<GlobalStreamId,List<FilterBinding>> bindings;
 public void prepare(Map stormConf, TopologyContext context,
      OutputCollector collector) {
```

```
    this.collector = collector;

    bindings = new HashMap<GlobalStreamId,List<FilterBinding>>();
    for(GlobalStreamId id : context.getThisSources().keySet()) {
      Fields fromId = context.getComponentOutputFields(id);
      ArrayList<FilterBinding> bounds =
       new ArrayList<FilterBinding>();
      for(FilterDefinition def : filters) {
        if(fromId.contains(def.fieldName)) {
          FilterBinding bind = new FilterBinding();
          bind.filter = def;
          bind.fieldNdx = fromId.fieldIndex(def.fieldName);
          bounds.add(bind);
        }
      }
      if(bounds.size() > 0) bindings.put(id, bounds);
    }
  }
```

理想情况下，在prepare阶段所做的确定输入字段的分析本来也可以用程序确定输出字段。可惜，declareOutputFields方法是在创建拓扑时调用，而不是在初始化拓扑时调用。因此，只有当已经定义了过滤器，每个过滤器包含的元组元素才会被指定，并直接由declareOutputFields返回：

```
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    for(FilterDefinition def : filters)
      declarer.declareStream(def.stream, def.fields);
  }
```

过滤器自身的实现使用了prepare方法中建立的过滤器绑定。传到bolt中的每个Tuple都带有关于该Tuple起源的元数据，其中包括GlobalStreamId。我们用getSource-GlobalStreamId来获得它，并用它来查找与该Tuple绑定的过滤器。如果找到对应的过滤器，就将该元组应用到列表中的所有过滤器，并将其发送到匹配的流中。对元组使用select方法，来提取与所配置的Fields元素匹配的元组：

```
public void execute(Tuple input) {
    List<FilterBinding> bound
      = bindings.get(input.getSourceGlobalStreamid());
    if(bound != null && bound.size() > 0) {
      for(FilterBinding b : bound) {
        if(b.filter.regexp.matcher(
            input.getString(b.fieldNdx)
          ).matches()) {
          collector.emit(b.filter.stream,
            input.select(b.filter.fields));
        }
      }
    }
  collector.ack(input);
  }
```

5.2.6.2 Basic bolt

如果一个bolt满足一些特定条件，那也可以将其实现为BasicBolt。这种bolt去掉了实现RichBolt时的一些样板代码。BasicBolt类的实现没有prepare步骤。它假设所需要的唯一的非序列化配置是获取collector对象。

BasicBolt还假定所有的元组都会用ack确认。这在Storm的深层代码中，而不是BasicOutputCollector中实现。

日志记录bolt

日志记录bolt是一个非常简单的bolt实现，它在测试拓扑时很有用。它所做的全部工作就是将输入元组的结果输出到控制台。这在测试拓扑时会很有用，用BasicBolt实现起来也很容易。这个bolt并不产生数据，只需要假定Tuple实现了正确的toString方法：

```java
public class LoggerBolt extends BaseBasicBolt {

  private static final Logger LOG
   = Logger.getLogger(LoggerBolt.class);
  private static final long serialVersionUID = 1L;

  public void execute(Tuple input, BasicOutputCollector collector) {
    LOG.info(input.toString());
  }

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

}
```

5.2.7 实现并使用spout

spout是一种特殊形式的Storm拓扑元素，它负责从外部获取数据。与bolt不同，它只需要实现一个名为IRichSpout的接口，并且不需要继承基类。

创建spout时，首先要调用open方法。这让spout可以配置到外部的连接，例如到队列或数据移动服务器的连接。与bolt类似，spout应该通过调用open方法获得SpoutOutputCollector对象，以供后面使用：

```java
public class EmptySpout implements IRichSpout {

  transient SpoutOutputCollector collector;
  public void open(Map conf, TopologyContext context,
      SpoutOutputCollector collector) {
    this.collector = collector;
  }
```

与bolt类似，spout也定义输出流。有一个常用的默认流，可以通过调用declare方法来定义。此外，还可以通过调用declareStream方法来定义命名流，这也与bolt类似：

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("first","second","third"));
  declarer.declareStream("errors",new Fields("error"));
}
```

根据spout的定义，它是基于“拉”（pull-based）的轮询接口。Storm基础架构会不断地调用nextTuple方法，该方法与bolt的execute方法类似：

```
public void nextTuple() {
  Utils.sleep(100);
  collector.emit(new Values("one","two","three"));
}
```

由于Storm使用轮询接口，当没有数据要发送时，建议你加入一条sleep命令。这有助于在拓扑空转时降低系统负载。在上面这个简单例子中，设置一个小的延迟也是出于同样的目的。

除了启动，拓扑也可以关闭、暂停和恢复。这多半会影响spout的处理。spout也有相应的方法可以用来启动和停止轮询进程。此外，当拓扑关闭时，还可以用close方法来干净地关闭消费者接口：

```
public void close() {
}

public void activate() {
}

public void deactivate() {
}
```

最后，拓扑可以用适当的spout来实现事务语义。当bolt对Tuple对象执行ack或fail命令时，该事件就会被传送到spout，由它决定对该元组的处理是进行提交还是重新处理：

```
public void ack(Object msgId) {
}

public void fail(Object msgId) {
}
```

实际情况中，这些事件是旧版本Storm的遗留问题。对于新的事务拓扑来说，建议使用Storm 0.8.0之后支持的Trident接口。该接口提供了一个在拓扑中实现事务语义的更干净的机制。本章稍后会介绍该接口。

上述方法之外，只需要再实现一个方法，那就是getComponentConfiguration方法。它用于spout的额外配置，一般是不会使用的，可以直接返回null：

```
public Map<String, Object> getComponentConfiguration() {
  return null;
}
```

“lorem ipsum”spout

尽管Storm包含若干拓扑测试机制（这些机制后面我们会讲到），但让spout可以生成随机数据往往也很有用。

“lorem ipsum”是著名的无意义文本，常被设计者用来模拟文本块的布局。该文本约一个段落长度，是混乱的拉丁文文本。现代版本的“lorem ipsum”可能会生成任意长度的文本。

Maven中央资源库中有一个Java类LoremIpsum，它可以根据下列初始段生成任意长度的字符串。

```
<dependency>
  <groupId>de.sven-jacobs</groupId>
  <artifactId>loremipsum</artifactId>
  <version>1.0</version>
</dependency>
```

首先，用要生成的元组的字段名来定义spout：

```
public class LoremIpsumSpout implements IRichSpout {

  private static final long serialVersionUID = 1L;
  String[] fields;

  public LoremIpsumSpout tuple(String...fields) {
    this.fields = fields;
    return this;
  }

  int maxWords = 25;
  public LoremIpsumSpout maxWords(int maxWords) {
    this.maxWords = maxWords;
    return this;
  }
```

当调用spout的open方法时，临时的LoremIpsum类就被实例化，因为它没有实现Serializable，故而无法在拓扑启动之前配置。像以前一样，仍然要获取collector元素以备后用：

```
transient LoremIpsum           ipsum;
transient Random               rng;
transient SpoutOutputCollector collector;
public void open(Map conf, TopologyContext context,
  SpoutOutputCollector collector) {
  this.collector = collector;
  ipsum = new LoremIpsum();
  rng   = new Random();
}
```

由于该spout完全是自包含的，因此事务方法（ack和fail）和生命周期管理方法（activate、deactivate和close）都可以留作存根（stub）实现。

为了减轻测试时的负载，nextTuple方法休眠一小段时间，然后发送包含不定长度的“lorem ipsum”文本的元组：

```java
public void nextTuple() {
  Utils.sleep(100);
  ArrayList<Object> out = new ArrayList<Object>();
  for(String s : fields)
    out.add(ipsum.getWords(rng.nextInt(maxWords)));
  collector.emit(out);
}
```

为了测试该spout类，我们使用一个简单的LocalCluster来查看其执行结果。首先构造一个包含两个LoremIpsum的拓扑：

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout1", new LoremIpsumSpout()
  .tuple("first","second","third"));
builder.setSpout("spout2", new LoremIpsumSpout()
  .tuple("second","third","forth"));
```

然后，为了给该拓扑一点事情做，我们向它加入上一节介绍的FilterBolt，让它过滤“lorem ipsum”文本两个众所周知的部分。注意该过滤器可以从两个spout中获得元组：

```java
builder.setBolt("filter", new FilterBolt()
  .filter("ipsum", "first", ".*ipsum.*","first")
  .filter("amet","second",".*amet.*","second")
).shuffleGrouping("spout1")
    .shuffleGrouping("spout2")
   ;
```

为了查看过滤的效果，我们使用LoggerBolt将输出显示到终端。它还识别出源流，这说明过滤成功：

```java
builder.setBolt("print", new LoggerBolt())
.shuffleGrouping("filter","amet")
.shuffleGrouping("filter","ipsum");
```

随后将拓扑提交给LocalCluster执行：

```
Config conf = new Config();

LocalCluster cluster = new LocalCluster();
cluster.submitTopology("example", conf, builder.createTopology());
```

最后，让主类休眠一段时间。这是因为LocalCluster运行的是内嵌的拓扑，多数工作都是在后台线程中完成的。

```
//Sleep for a minute
Thread.sleep(60000);
```

运行拓扑之后，输出信息应如下所示：

```
6206 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
  - source: filter:2,
  stream: ipsum, id: {},
  [Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
  sed diam nonumy eirmod tempor invidunt ut]
6306 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt

  sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
  labore et dolore magna aliquyam]
6407 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
  - source: filter:2,
  stream: ipsum, id: {}, [Lorem ipsum dolor sit amet,
  consetetur sadipscing elitr, sed diam nonumy eirmod tempor
  invidunt ut labore et dolore magna]
6407 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
  - source: filter:2,
  stream: amet, id: {}, [Lorem ipsum dolor sit amet,
  consetetur sadipscing elitr, sed diam nonumy eirmod tempor
  invidunt ut labore et dolore magna aliquyam]
```

5.2.7.1　将Storm与Kafka连接

随着Kafka 0.8的引入，Kafka与Storm 0.9的集成就一直处于不断变动的状态。storm-contrib库中的storm-kafka spout实现目前支持Kafka 0.7.2版本。可以从另一个称为storm-kafka-0.8-plus（http://github.com/wurstmeister/storm-kafka-0.8-plus ）的项目中获得能与Kafka 0.8集成的

Kafka spout版本。也可以从clojars.org获得该项目，其方式与在Maven的pom.xml中包含Storm相同，使用下列工件就可以将该Kafka spout包含到项目中：

```
<dependency>
  <groupId>net.wurstmeister.storm</groupId>
  <artifactId>storm-kafka-0.8-plus</artifactId>
  <version>0.2.0</version>
</dependency>
```

要使用spout，首先要配置KafkaConfig对象，将其指向相应的代理集合和所需的主题。例如，下列代码从ZooKeeper获取代理列表。然后就用Storm的消费者组将自己与wikipedia-raw主题关联：

```
BrokerHosts      hosts    = new ZkHosts("localhost");
SpoutConfig      config   = new SpoutConfig(hosts,
  "wikipedia-raw", "", "storm");
config.scheme = new SchemeAsMultiScheme(new StringScheme());
```

Config.scheme告诉spout将输入消息解释为字符串，而非其他格式。如果要解释为其他格式，可能需要实现针对该格式的模式。

要使用该拓扑，首先创建KafkaSpout，然后就像使用其他任何spout一样使用该spout。这个简单的测试拓扑负责报告Wikipedia-raw主题所产生的事件：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("kafka", new KafkaSpout(config), 10);
builder.setBolt("print", new LoggerBolt())
  .shuffleGrouping("kafka");

Config conf = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("example", conf, builder.createTopology());
//Sleep for a minute
Thread.sleep(60000);
```

5.2.7.2　将Storm与Flume连接

Storm与Flume之间存在根本上的“阻抗失谐”（impedance mismatch）问题。Storm是轮询式的消费者，假定的是“拉”模式。Flume基本的sink模型是基于“推”的。尽管一直有人在尝试解决这个问题，但还没有可以令人接受的机制将Flume和Storm直接连接起来。

对于这个问题有一些解决方案，但都需要加入额外的队列系统或数据移动系统。“坊间”流传着两种基本途径：第一种是使用兼容高级消息队列协议（Advanced Message Queuing Protocol，AMQP）的队列系统。对于Flume应用来说，最流行的是RabbitMQ队列系统，因为有一个由Kenshoo开发的Flume sink（http://github.com/kenshoo/flume-rabbitmq-sink）。然后就可以用Storm的AMQPSpout从RabbitMQ读取消息。

第二种是对Flume使用Kafka sink，然后按照上一节所讲的方法将Storm与Kafka集成。这时使用Flume的真正原因只有一个，那就是与已有的基础架构集成。你可以在https://github.com/baniuyao/flume-ng-kafka-sink 找到Flume/Kafka sink。

5.2.8　分布式远程过程调用

除了开发标准的处理拓扑来从流中消费数据，Storm还具备实现分布式远程过程调用（Distributed Remote Procedure Call，DRPC）的功能，用于简单地实现分布式处理服务，例如进行大量昂贵的计算。通过使用spout和bolt的特殊组合，这些拓扑就能实现跨多台机器运行的复杂过程。

严格来说，这并不是实时流分析场景中通常的做法，但肯定可以用来扩展实时分析流水线的某些环节。它也可以用来建立基于组件的服务，从而使环境更易于管理。

5.2.8.1 DRPC服务器

DRPC请求是由独立的Storm服务器管理的，该服务器负责管理DRPC客户端和为DRPC请求提供服务的Storm拓扑之间的通信。在分布式环境下，用Storm命令行工具来启动该服务器：

```
$ ./bin/storm drpc
```

如果处于测试环境，可以启动本地的DRPC服务器。这与用于测试的LocalCluster类似：

```
TopologyBuilder builder = new TopologyBuilder();
LocalDRPC drpc = new LocalDRPC();
```

5.2.8.2 构造DRPC拓扑

DRPC拓扑与其他所有Storm拓扑类似，依赖于通过spout和bolt的特殊组合来完成DRPC功能。拓扑通过spout获得单个String值作为输入，再通过ReturnResults bolt返回单个String值。例如，下列拓扑实现了“幂”函数，其输入为以逗号间隔的双精度数值列表，第一个数值为底数，第二个数值为指数。其返回值是表示该幂输出值的字符串：

```
//Omit the drpc argument when submitting to the cluster
builder.setSpout("drpc", new DRPCSpout("power", drpc));
builder.setBolt("split", new SplitBolt()).shuffleGrouping("drpc");
builder.setBolt("power", new PowerBolt()).shuffleGrouping("split");
builder.setBolt("return", new ReturnResults()).shuffleGrouping("power");
```

SplitBolt将输入的以逗号间隔的列表分成两个Double值。它还传递元组的return-info元素，该元素被ReturnResults对象用来向DRPC服务器提交结果：

```
public class SplitBolt extends BaseBasicBolt {

  private static final long serialVersionUID = -3681655596236684743L;
  public void execute(Tuple input, BasicOutputCollector collector) {
    String[] p = input.getString(0).split(",");
    collector.emit(
      new Values(
        Double.parseDouble(p[0]),
        Double.parseDouble(p[1]),
        input.getValue(1)
      )
    );
  }

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("x","y","return-info"));
  }
}
```

PowerBolt实现以x和y值为输入，计算幂函数值。其返回值为DRPC幂函数需要的String类型。它还必须传输return-info对象：

```
public class PowerBolt extends BaseBasicBolt {

  private static final long serialVersionUID = 1L;

  public void execute(Tuple input, BasicOutputCollector collector) {
    collector.emit(
      new Values(
        ""+Math.pow(input.getDouble(0),
        input.getDouble(1)),
        input.getValue(2)
      )
    );
  }

  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("result","return-info"));
  }
}
```

多数情况下，这样的拓扑都不是手工创建的。在旧版本的Storm中，使用LinearDRPC-TopologyBuilder类来构造这些拓扑。这个类会向拓扑中自动地添加相应的DRPCSpout spout和ReturnResults bolt，并确保拓扑本身是线性的。在新版本的Storm中，该类被弃用，取而代之的是它的Trident版本，后者是用newDRPCStream方法而不是惯用的newStream方法来创建的。我们会在下一节中详细讨论Trident这种特定领域语言。

5.2.9　Trident：Storm的DSL

在0.8.0版本中，Storm引入了一种新的特定领域语言，目标是使其成为Storm拓扑开发的首选方法。这种语言称为Trident。在Storm 0.9.0中对Trident进行了深入的开发工作。

Trident接口的目标是提供类似于Hadoop框架中的Cascading那样的高层抽象。Trident直接对流进行操作，而不是提供非常原始的spout和bolt接口。可以用高层概念来操作这些流，例如连接（join）、分组（group）、聚集（aggregate）和过滤（filter）。它还加入了一些状态管理和持久化原语，特别是在使用聚集的情况下。

在底层，Trident仍然使用同样的bolt和spout接口。所不同的是，Trident拓扑定义的操作并不一定都会生成新的bolt。这使得拓扑可以通过将几个操作合并成一个物理bolt并避免bolt之间的通信开销来获得更高的性能。

本节我们介绍怎样使用Trident的高层接口来构造Storm拓扑。

5.2.9.1 Trident流和spout

Trident拓扑通常是在Stream对象上进行操作的。Trident中的流在概念与常规的Storm拓扑中的spout或bolt所定义的流很相似。它们的差别在于，前者的结构是在拓扑定义中定义和修改的，后者则属于bolt的内在部分。

在拓扑中定义流的结构对于编写拓扑有两个好处。首先，将元组结构从业务逻辑中分离出来，编写可重用的组件更容易。其次，拓扑可读性要好得多。因为数据流程和数据定义在同一个地方，对处理过程的观察就更容易。而在“传统”的拓扑中，如果确定每个流是怎么定义的，就要分别查看每个bolt。除了上述两个优点，这种做法也有缺点：使用Trident语言来表达计算要比显式地定义拓扑更难一些。不过，这往往只是个人喜好问题。

Trident拓扑中的流是在spout的基础上构建的。这些spout可以是标准拓扑中同样的IRichSpout实现，或者是某种更专门的Trident spout。如果使用基本spout，例如LoremIpsumSpout，那么很简单，就是在TridentTopology类中使用newStream来定义一个流：

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout());
```

5.2.9.2 局部操作：Filter和Function

构造拓扑时，Trident会将某些操作视为“分区局部”操作。“分区局部”的含义是这类操作可以在单个bolt局部链式执行。这样Trident就可以利用更多有效的数据结构来减少所需要的通信量。

最基本的局部操作是Filter，它通过继承BaseFilter类来实现。Filter类要实现一个方法isKeep，用来决定是否继续处理某个元组。例如，下列Filter实现以随机选择的方式保留一半的结果：

```
public class RandomFilter extends BaseFilter {
  private static final long serialVersionUID = 1L;
  Random rng = new Random();
  public boolean isKeep(TridentTuple tuple) {
    return rng.nextFloat() > 0.5;
  }
}
```

下一个简单的操作是Function，它通过继承BaseFunction类来实现（有些Trident教程错误地将Filter写成BaseFunction的子类，但在0.9.0版本中并非如此）。

Function对象在功能上与“传统”Storm拓扑中的BasicBolt对象相近。它们以单个Tuple为输入，可以生成0个或多个元组，附加到初始元组的后面。如果没有生成任何元组，初始元组就会从流中被过滤掉。例如，下列Function实现向每个输入Tuple加上一个随机数字：

```
public class RandomFunction extends BaseFunction {
  private static final long serialVersionUID = 1L;
  Random rng = new Random();
  public void execute(TridentTuple tuple, TridentCollector collector) {
    collector.emit(new Values(rng.nextDouble()));
  }
}
```

Filter和Function类都使用each方法来将自己加入流中。对于Filter来说，each方法的使用形式是each（inputFields，filter）：

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
  .each(new Fields(), new RandomFilter())
;
```

注意Fields构造器是空的。这是因为RandomFilter的过滤并不依赖于任何字段。它实际上并不会修改经过它处理的Tuple，因此这些Tuple仍然包含“first”、“second”和“third”元素。

Function的each方法的参数与之类似，通常有两种形式。第一种是each（function，outputFields），这是RandomFunction实现使用的形式：

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
  .each(new RandomFunction(),new Fields("x"))
;
```

第二种形式允许函数有输入字段，这与filter操作类似。例如，SquareFunction实现只有一个参数：

```
public class SquareFunction extends BaseFunction {
  private static final long serialVersionUID = 1L;
  public void execute(TridentTuple tuple, TridentCollector collector) {
    collector.emit(new Values(tuple.getDouble(0)*tuple.getDouble(0)));
  }
}
```

当使用RandomFunction将其放到拓扑中时，它会以x字段作为输入，生成的y字段作为输出，并且将输出附加到元组后面：

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
.each(new RandomFunction(),new Fields("x"))
.each(new Fields("x"), new SquareFunction(),new Fields("y"))
.each(new Fields("x","y"),new PrintFunction(), new Fields())
;
```

该拓扑运行一段时间之后，应该会有下列输出（简洁起见，我们只将x和y字段放入PrintFunction）：

```
[0.8185738974522312, 0.6700632255901359]
[0.04147059259035091, 0.0017198100497948677]
[0.512209146992401 7, 0.262358210262683 8]
[0.03233058140451927 6, 0.0010452664939542477]
[0.4988177749199945 7, 0.24881917257613437]
[0.914029629250604 3, 0.835450163147997 1]
[0.807521215873791, 0.6520905140862857]
[0.03596640476063595, 0.0012935822714058964]
[0.7539011202358764, 0.5683668990929094]
```

5.2.9.3 重新分区操作

在Trident中，将分组操作称为分区操作。但从本质上来说，分区操作与标准拓扑中的分组操作没什么两样。分区操作也意味着用网络传输元组，这与在标准拓扑中的分组操作一样：

·Shuffle分区操作将元组均匀地分配给目标分区。

·Broadcast分区操作将每个元组发送给所有目标分区。

·PartitionBy分区操作等价于fieldGrouping。它以每个元组中的字段列表为输入，使用这些字段的哈希值来确定目标分区。

·Global分区操作将所有元组发送到同一个分区。Trident还增加了一个名为batchGlobal的变种，它将属于同一个批次的所有元组发送到一个分区

中。不同批次的元组则可能发送到不同分区。

·还有一个partition函数，它与CustomStreamGrouping类似，用来实现自定义的分区方法。

除了这些分区方法，Trident还实现名为groupBy的特殊类型的分区操作。这种分区操作的功能与Map-Reduce作业中的reducer阶段非常相似，通常是Trident中最常用的分区。

GroupBy操作首先根据它指定的字段来实行partitionBy分区操作。随后将每个分区中具有相同哈希值的值归为一组，以在GroupedStream中进一步处理。这之后就可以直接对这些组应用聚集操作。

5.2.9.4　聚集计算

Trident有两种流上的聚集计算方法：aggregate和persistent-Aggregate。如果一个流的函数有效地实现了ReducerAggregator或Aggregator聚集器，对该流应用aggregate方法就会对数据进行global分区，这将使得所有元组被发送到相同的分区，从而逐批聚集。CombinerAggreator类型的聚集计算方法首先对每个分区计算聚集的中间结果，然后对这些中间结果执行global分区操作。

内置的聚集器Count和Sum都属于CombinerAggreator。其他自定义的聚集计算方法可以通过继承BaseAggregator来实现。下一节给出了一个例子，这个例子实现了分区局部的聚集计算。

另外一种聚集计算方法persistentAggregator与aggregate类似，唯一

的区别在于，前者将自己的输出记录到State对象中。这些State对象经常与Trident拓扑中的spout一起使用，用来确保事务语义等特性。

PersistentAggregate方法返回的是TridentState对象，而不是Stream对象。每当键的状态发生变化，TridentState对象的方法newValueStream就会发送一个元组。如果要从使用本地内存作为后备存储的聚集操作（例如MemoryMapState类）中提取结果，这会很有用。

目前，MemoryMapState类是Trident唯一的内置State。它主要用于测试目的，而其他State对象（例如memcached）则会保存到更持久的存储中。

5.2.9.5　分区局部的聚集计算操作

尽管常规的聚集计算操作隐含了全局分区事件，但是partitionAggregate方法还是支持分区操作：

```
public class KeyDoubleAggregator
  extends BaseAggregator<Map<Object, Double>> {

  private static final long serialVersionUID = 1L;
  public Map<Object, Double> init(Object batchId,
    TridentCollector collector) {
    return new HashMap<Object,Double>();
  }

  public void aggregate(Map<Object, Double> val, TridentTuple tuple,
    TridentCollector collector) {
    Object key = tuple.get(0);
    if(val.containsKey(key))
      val.put(key, val.get(key) + tuple.getDouble(1));
    else
      val.put(key, tuple.getDouble(1));
  }

  public void complete(Map<Object, Double> val,
    TridentCollector collector) {
    for(Entry<Object,Double> e : val.entrySet()) {
      collector.emit(new Values(e.getKey(),e.getValue()));
    }
  }
}
```

经典的“单词计数”例子

“单词计数”是大数据处理的“Hello World”，任何对实时流处理的讨论都不会完全忽略它。

这个例子使用从Kafka输入到Trident的维基百科文章流，来实现单词计数示例。该数据源实际上是由本章下一部分讨论的Samza包提供的，它提供了一个便于使用的测试数据源。用TransactionalTridentKafkaSpout类就可以对它进行配置：

```java
TridentTopology topology = new TridentTopology();

TridentKafkaConfig      config    = new TridentKafkaConfig(
  new ZkHosts("localhost"),
  "wikipedia-raw",
  "storm"
);
config.scheme = new SchemeAsMultiScheme(new StringScheme());

topology.newStream("kafka",
  new TransactionalTridentKafkaSpout(config)).shuffle()
```

该spout发送一个JSON对象的字符串，必须首先解析该字符串，然后将其分解为单词供进一步处理。简单起见，该函数只处理原始输出的标题部分：

```java
.each(new Fields("str"),new Function() {

  private static final long serialVersionUID = 1L;

  transient JSONParser parser;
  public void prepare(Map conf, TridentOperationContext context) {
    parser = new JSONParser();
  }
  public void cleanup() { }

  public void execute(TridentTuple tuple,
    TridentCollector collector) {
    if(tuple.size() == 0) return;
    try {
      Object obj = parser.parse(tuple.getString(0));
      if(obj instanceof JSONObject) {
        JSONObject json = (JSONObject)obj;
        String raw = (String)json.get("raw");
        raw = raw.substring(2,raw.indexOf("]]"));
        for(String word : raw.split("\\s+")) {
          collector.emit(new Values(word));
        }
      }

    } catch (ParseException e) {
      collector.reportError(e);
    }
  }
}, new Fields("word")).groupBy(new Fields("word"))
```

这个函数的输出是每个单词的元组。通过对单词分组，然后应用persistent-Aggregate，就可以获得一段时间内每个单词的计数。每当输出

中出现一个单词，该单词的聚集量就会加上它在标题中出现的次数，然后就进行聚集计算：

```
.persistentAggregate(
  new MemoryMapState.Factory(),
  new Count(),
  new Fields("count")
)
.newValuesStream()
```

最后，将这些值读入Debug函数，这个函数与本章前面实现的LoggerBolt非常相似：

```
.each(new Fields("word","count"), new Debug("written"));
```

拓扑运行一段时间之后，在控制台上会出现下列输出：

```
DEBUG(written): [Category:Water, 1]
DEBUG(written): [Fleurimont, 1]
DEBUG(written): [for, 1]
DEBUG(written): [creation/Orin, 1]

DEBUG(written): [Wikipedia, 1]
DEBUG(written): [talk:Articles, 1]
DEBUG(written): [of, 5]
DEBUG(written): [players, 2]
DEBUG(written): [F.C., 1]
DEBUG(written): [List, 4]
DEBUG(written): [Arsenal, 1]
DEBUG(written): [Colby, 1]
DEBUG(written): [Jamie, 1]
DEBUG(written): [Special:Log/abusefilter, 1]
DEBUG(written): [Special:Log/abusefilter, 2]
```

5.3 用Samza处理数据

源于LinkedIn的Samza项目是实时处理领域的新成员。Samza最近才开源并加入到Apache孵化器家族中，它是建立在Apache YARN基础架构之上的实时处理框架。该项目本身仍然处于早期阶段，与已经有多年历史的Storm相比更显稚嫩。不过，在实时处理领域，Samza已经有了一些用武之地。

本节介绍Samza的基础架构及其初步使用方法。首先概述Apache YARN，它是Samza的服务器基础架构，替代了Storm的nimbus/supervisor服务器的位置（实际上Storm也可以在Apache YARN上运行，通过使用Yahoo！的Storm-YARN项目https://github.com/yahoo/storm-yarn 就可以做到）。接下来介绍Samza应用本身。与Storm的Trident系统类似，Samza提供一些原语来构建常见类型的流应用，并在流处理应用内部维护状态。

5.3.1 Apache YARN

Samza并不自行实现服务器管理框架，而是将多数的系统基础架构下移到Apache YARN上。YARN是Yet Another Resource Negotiator的缩写，它负责管理Samza处理流水线的部署、容错和安全。

5.3.1.1 背景

YARN最初是为了克服Hadoop项目的局限性而诞生的。Hadoop项目

的核心构建在JobTracker服务器之上，该服务器负责管理向其他运行TaskTracker的服务器分发任务、mapper、reducer。想要提交作业的客户端必须要连接JobTracker，并指定输入集合，这个集合通常是驻留于Hadoop分布式文件系统中的分布式数据块集合，以及需要分发到每个节点的所有支撑代码或数据。JobTracker然后就将该请求分解为小的任务，并在tracker上对这些任务进行调度，如图5-4所示。

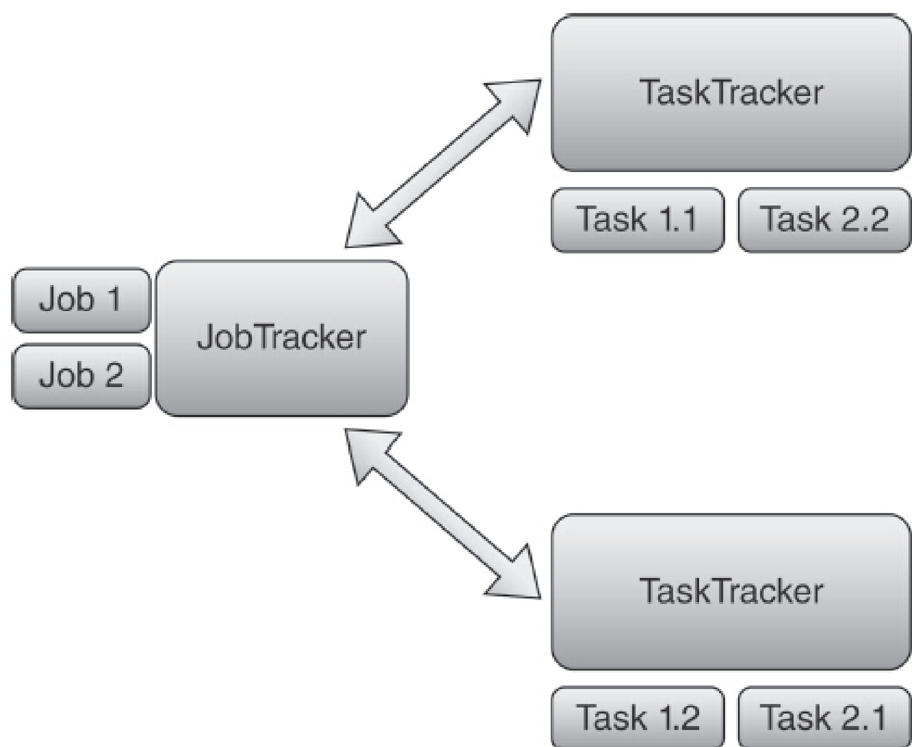


图　5-4

这在适度规模的集群上运行良好，但在5000个左右的多核服务器组成的集群上运行就有些实际的限制。在任务总数上也存在限制（不论是单个

作业中的任务还是分布在多个作业中的任务），这是因为所有任务的相关信息都要保留在JobTracker自己的内存中。

随着物理硬件不断扩展，现代数据中心可以容纳非常大的集群，这些规模上的限制开始对Hadoop的可扩展性造成负面影响。此外，类数据库应用和长时流处理应用这些新的流处理负载，它们的假设场景是有长时间运行但少量的reducer任务，同时又有大量的短暂运行的mapper任务，这与Hadoop处理模型有点不相匹配。

为了应对不断增长的集群以及不断变化的负载这两方面的需求，就开发了第2代Hadoop：YARN。

5.3.1.2 架构

从理论上说，YARN架构与原始的Hadoop基础架构没有太大差异。YARN的高层服务器是ResourceManager和NodeManager，而不是Hadoop的JobTracker和TaskTracker。一个很重要的差异在于，YARN的高层服务器现在管理的是应用，而不是单独的任务。

YARN中的应用由一个ApplicationMaster和许多容器组成，这些容器驻留在每个节点上。顾名思义，ApplicationMaster负责协调作业，并管理归属它的容器，这些容器里运行着各个任务。这些组件之间的关系如图5-5所示。

在Hadoop 2的Map-Reduce实现中，ApplicationMaster担负了JobTracker的角色。唯一的不同是每个ApplicationMaster只管理单个作业的任务，而不是大量的作业。这就使得每个ApplicationMaster可以在

Map-Reduce环境下独立运作，并且使更加精确的资源管理和安全模型成为可能，因为此时作业基本上是完全独立的。

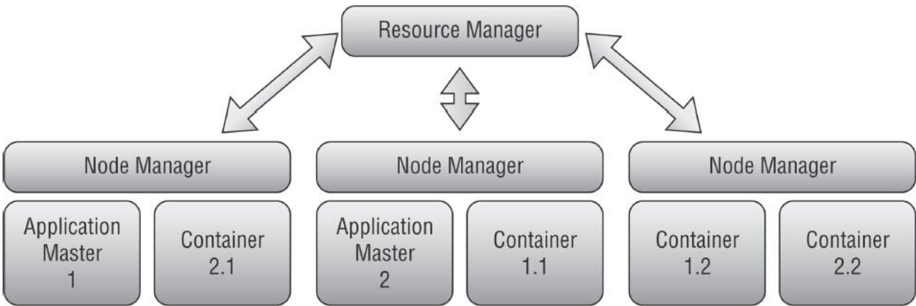


图　5-5

5.3.1.3　与Samza的关系

Samza是建立在YARN之上的应用。Samza应用拥有所需要的ApplicationManager，用来管理在YARN的Container中驻留的Samza的TaskRunners。TaskRunners运行StreamTasks，后者在Samza中的作用相当于Storm的Bolt。

Samza所有的通信都是通过Kafka的代理来进行的。与Hadoop Map-Reduce应用中的DataNode类似，这些代理与Samza容器通常位于相同的机器上。这样，Samza就使用Kafka的主题和自然分区来实现流处理应用的分组特性。

5.3.2　从YARN和Samza开始

Hadoop 2的发布已经有段时间了，但它在生产环境中还是不太常见，

不过这种情况正在迅速改变。对于许多用户来说最重要的是，目前Amazon的Elaztic MapReduce产品已经支持Hadoop 2作为正式版本，这样启动集群就很容易了。

目前至少已经有两个主流的Hadoop发行版支持Apache YARN，更多的发行版正源源不断地加入。使用它们自己的集群管理工具来启动YARN集群是很简单的。唯一的缺点是打包的发行版往往会有一些随意的补丁，并且某些版本可能会滞后于Apache项目最新发布的版本。

此外，可以使用Apache包启动单个节点集群供测试使用，也可以在多个节点上启动分布式集群。

5.3.2.1 单节点Samza

单个节点上使用Samza的最简单的方式就是用Hello Samza项目中的单节点YARN安装包。该项目包括了Samza、ZooKeeper、Kafka和YARN。所有这些都打包在一起，便于进行开发工作。

开始时，首先从Github检出hello-samza Git资源库。该资源库包括一个脚本grid，该脚本负责构建和安装一个完整的Samza：

```
$ git clone https://github.com/apache/incubator-samza-hello-samza.git
$ cd incubator-samza-hello-samza/
$ ./bin/grid bootstrap
```

JAVA_HOME NOT SET

在某些系统（例如Mac OS X）中，当运行grid时，会返回JAVA_HOME not set错误：

```
$ ./bin/grid bootstrap
JAVA_HOME not set. Exiting.
```

在OS X中，下列命令可以将环境变量JAVA_HOME设置为正确值：

```
$ export JAVA_HOME=$(/usr/libexec/java_home)
```

第一次运行时，grid脚本会构建和安装Samza到本地的Maven资源库。这样就不需要像Samza网站上的教程那样单独检出和构建git资源库。grid还会下载并启动支持Samza的对应版本的ZooKeeper和Kafka。构建Samza需要3到5分钟，安装ZooKeeper和Kafka则需要更长时间，具体时间取决于网络速度。grid应用还会下载和安装单节点版本的YARN。这个安装包大约有100MB，因此第一次安装Hello Samza项目时，最好有稳定的网络连接。

如果一切顺利，在浏览器中输入http://localhost：8088，显示的网页应该如图5-6所示。

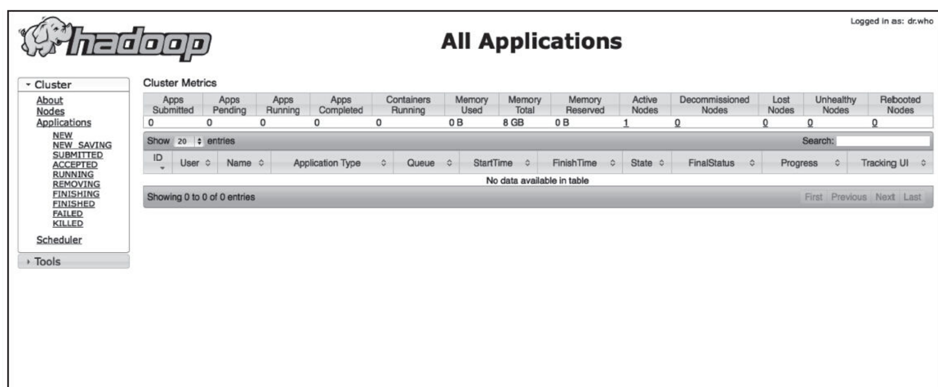


图 5-6

如果要关闭Samza，可以用grid脚本提供的命令以正确的顺序终止所

有进程：

```
$ ./bin/grid stop all
```

如果要再次启动gird，只需要使用start命令：

```
$ ./bin/grid start all
```

如果某个进程已经在别的服务器池中运行（例如ZooKeeper），重启gird时就可以忽略掉该进程。这时，就不要用all命令，而应该指定要启动的各服务器的名称。

Hello Samza项目还包含了一些可以运行的示例代码。本章中的一些例子使用了第一部分示例代码。该Samza作业从维基百科IRC服务器中读取维基百科的文章流，并将其以JSON格式发布到本地Kafka集群的wikipedia-raw主题中。要在grid中启动该作业，需要运行下列代码：

```
$ ./deploy/samza/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/samza/config/wikipedia-feed.properties
```

该作业启动后，使用Kafka自带的控制台消费者来检查文章是否发布到了Kafka中：

```
$ ./deploy/kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost:2181 --topic wikipedia-raw
```

5.3.2.2　多节点Samza

除了不需要HDFS（除非其他应用会用到），在多节点环境配置Samza与建立Hadoop 2环境非常相似。

首先，确保ZooKeeper集群可用。Samza和Kafka代理都需要ZooKeeper。如果对ZooKeeper集群安装和配置有什么问题，可以参考第3章。

然后，在YARN grid用到的机器上（除了用作ResourceManager的机器之外）安装和配置Kafka代理。Samza要频繁使用Kafka来通信，因此Kafka的代理通常与组成Samza YARN grid的NodeManagers位于相同的机器上。对于Kafka的安装和配置如有疑问，请参考第4章。

如果ZooKeeper和Kafka都安装完毕，那就该建立YARN集群了。这一节中，我们假设你是用Apache的构建包而不是商业发行版来构造YARN集群的。我们还假设所使用的操作系统对应的软件包管理器没有合适的Hadoop 2.2.0软件包（许多仍然有更老的Hadoop 1.0.3软件包）。

首先，下载并解压Apache二进制安装包到每个机器上。编写本书时它的版本号是2.2.0：

```
$ wget http://mirrors.sonic.net/
    apache/hadoop/common/hadoop-2.2.0/hadoop-2.2.0.tar.gz
$ cd /usr/local
$ sudo tar xvfz ~/hadoop-2.2.0.tar.gz
$ sudo ln -s hadoop-2.2.0/ hadoop
```

Apache YARN靠一系列环境变量来获知从哪里寻找各种软件包。有许多地方可以设置环境变量，但为了对所有用户生效，你应该在/etc/profile或/etc/profile.d/hadoop.sh中设置。多数系统中，/etc/profile自动包含了/etc/profile.d中的所有脚本。如果将YARN解压到/usr/local目录，/etc/profile.d/hadoop.sh文件的内容如下：

```
export YARN_HOME=/usr/local/hadoop
export HADOOP_MAPRED_HOME=$YARN_HOME
export HADOOP_COMMON_HOME=$YARN_HOME
export HADOOP_HDFS_HOME=$YARN_HOME
export PATH=${PATH}:${YARN_HOME}/bin:${YARN_HOME}/sbin
```

要看一下环境是否已经改变，那就退出然后重新登录，或者打开一个新的终端。然后，检查YARN_HOME环境变量，它的值应该是/usr/local/hadoop。检查Hadoop版本，应该返回2.2.0：

```
$ echo $YARN_HOME
/usr/local/hadoop
$ hadoop version
Hadoop 2.2.0
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
This command was run using
/usr/local/hadoop-2.2.0/share/hadoop/common/hadoop-common-2.2.0.jar
```

由于Samza使用该grid，因此没有必要配置HDFS。如果Map-Reduce负载还将使用该grid，那就参照相应的YARN文档来配置NameNode和DataNode服务器。唯一需要为Samza配置的就是/usr/local/hadoop/etc/hadoop中的yarn-site.xml（将resource-manager.mydomain.net替换为相应的主机名）。

```
<?xml version="1.0"?>
<configuration>
<property>
<name>yarn.resourcemanager.hostname</name>
<value>resource-manager.mydomain.net</value>
</property>
</configuration>
```

现在就可以在Kafka grid上启动ResourceManager和NodeManager。要启动Resource-Manager，只需要登录机器，使用yarn-daemon.sh来启动服务器：

```
$ yarn-daemon.sh -config $YARN_HOME/etc/hadoop start resourcemanager
```

然后在Samza grid的每个节点上以同样的方式启动NodeManager：

```
$ yarn-daemon.sh -config $YARN_HOME/etc/hadoop start nodemanager
```

默认情况下ResourceManager会在8088端口启动一个Web服务器。你可以通过查看该Web服务来确保每个节点都已经向资源管理器报到。这个时候，最常见的问题是防火墙配置不正确。

5.3.3　将Samza集成进数据流程

将Samza集成到已有的Kafka环境是很简单的，这是因为Samza所有的通信都使用Kafka。如果已经有一个代理集合负责处理生产负载，那就直接使用第4章中介绍的MirrorMaker来将所需要的主题镜像到Samza Kafka grid中。这样，Samza就可以从后者那里轻易地访问输入的主题。

另外一种方法是，在Kafka代理用来采集数据的机器上安装Samza grid。这种方法有一点运行上的缺陷，因为总是存在这样的可能：某个正在处理的作业可能会使机器锁死并停机。不过，这种方法通常运行起来更加高效，因为Kafka代理往往有空闲的处理周期。

5.3.4　Samza作业

当集群配置完毕，并且数据已经导入grid的Kafka部分时，就应该实现Samza作业（Job）了。Samza作业大致相当于Storm的Bolt。但是，Samza的作业不像Storm的Bolt那样在拓扑中组装在一起，它们是相互独立

的实体，它们之间的联系仅限于向同一个Kafka主题的读或写。

一方面，这样做可能会让资源的使用更高效，并且对联系紧密的流程的管理也更容易。例如，某个Samza作业可以负责获得一个输入流，并生成一个“有效”的输入流，供下游流的任意数量的作业使用。也容易加入新的进程来利用这些流。否则，即使处理方面的差别很小，这些进程也可能必须重新处理整个数据流。

这种方法的缺点是，作业拓扑的结构是完全抽象的。在Storm模型中，拓扑是一清二楚的，所有的处理步骤都由一个中心控制和监控。在Samza模型中，拓扑隐含在主题的排列组合中，永远不会被显式表述。在上游流的作业发生变化时，这可能导致管理方面的问题，或者无意间向拓扑结构中引入环路。

5.3.4.1　准备一个作业应用

Samza作业由两部分组成。第一部分是实际的代码，它是StreamTask接口的实现。第二部分是任务（Task）的配置，包括输入流的名称以及对日志、监控和其他辅助功能的配置。任何数量的作业实现都可以共存并打包在一起，以便于部署。

如果不使用前面介绍的grid实现，那就得安装Samza Maven包。在Maven Central或其他Maven资源库中还没有这些包，因此你需要在本地构建和安装。从Git资源库中检出Samza，然后用所提供的gradlew驱动来运行Gradle脚本：

```
$ git clone http://git-wip-us.apache.org/repos/asf/incubator-samza.git
$ cd incubator-samza
$ ./gradlew -PscalaVersion=2.8.1 clean publishToMavenLocal
```

脚本运行完成之后，应该修改包含作业实现的项目，使其pom.xml文件中包含samza-api依赖：

```
<dependency>
<groupId>org.apache.samza</groupId>
<artifactId>samza-api</artifactId>
<version>0.7.0</version>
</dependency>
```

5.3.4.2　配置作业

Samza作业的配置是通过使用属性文件完成的，在向YARN框架提交作业时该属性文件会被传递给Samza框架。开头是作业工厂类的说明和作业名称，以及发布包：

```
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=wordcount-split
yarn.package.path=

file://${basedir}/target/
${project.artifactId}-${pom.version}-dist.tar.gz
```

这个工厂类通常都是YarnJobFactory，作业名称这里设为wordcount-split，我们在下一节实现这个类。yarn.package.path是由构建进程填写的，它指定了YARN向每个节点传输的归档文件的名称。这些归档文件包含了所需的所有JAR文件以及实现作业的代码。

接下来定义任务。至少要定义task.class和task.inputs属性：

```
task.class=wiley.streaming.samza.WordSplitTask
task.inputs=kafka.wikipedia-raw
```

也可以包含任务的检查点特性的属性，该特性使用Kafka来记录任务状态，这样Samza就可以在发生故障时恢复：

```
task.checkpoint.factory=
org.apache.samza.checkpoint.kafka.KafkaCheckpointManagerFactory
task.checkpoint.system=kafka
task.checkpoint.replication.factor=1
```

然后就是Samza的度量报告系统。这完全是可选的，这里snapshot和jmx风格都可以使用。这个例子同时使用了这两种风格：

```
metrics.reporters=snapshot,jmx
metrics.reporter.snapshot.class=
org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory
metrics.reporter.snapshot.stream=kafka.metrics
metrics.reporter.jmx.class=
org.apache.samza.metrics.reporter.JmxReporterFactory
```

作业配置的序列化部分可以定义作业中使用的各种序列化器（Serializers）和反序列化器（Deserializers）（在处理领域的术语中这两者合称“SerDe”）。这个例子使用内置的JSON SerDe以及Samza所用的Metrics SerDe作为度量快照：

```
serializers.registry.json.class=
org.apache.samza.serializers.JsonSerdeFactory
serializers.registry.metrics.class=
org.apache.samza.serializers.MetricsSnapshotSerdeFactory
```

最后定义输入输出系统。这样就能在Samza环境中实现数据流。它们都是可插拔的系统，尽管目前Samza只自带了Kafka系统。下列配置定义task.input以及（在任务自身中定义的）输出所使用的Kafka系统。还用前面的度量定义规定了流系统使用的度量：

```
systems.kafka.samza.factory=
org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
# Normally, we'd set this much higher, but we want things to
# look snappy in the demo.
systems.kafka.producer.batch.num.messages=1
systems.kafka.streams.metrics.samza.msg.serde=metrics
```

应该根据部署环境修改systems.kafka.consumer.zookeeper.connect和kafka.producer.metadata.broker.list的值。如果使用Hello Samza grid，上面的值就是合适的。

5.3.4.3 任务通信

上一节的systems和serializer Job属性定义了要实现的任务的输入机制，并在一定程度上定义了输出机制。在Samza中这是通过SystemConsumer和SystemProducer接口实现的，这两个接口分别定义了任务的输入和输出。

SystemProducer是通过作业配置中的task.input属性来定义的，它负责轮询输入分区。它会生成IncomingMessageEnvelope对象，该对象随后会被传递给任务。它由三部分组成：

·键：可以通过getKey方法访问。这个键用来对Kafka数据流中的数据进行分区，直接对应了Kafka消息的key部分。

·消息：可以通过getMessage访问，通常包含流的载荷。

·StreamSystemPartition对象：可以通过getSystemStreamPartition方

法访问。它包含关于消息的元数据，例如最初的分区和偏移量信息。多数任务不需要直接访问该信息。

通常可以通过SystemStream来访问SystemProducer，前者被定义为任务类的一个静态的final成员。例如，要在主题“ouput”上定义Kafka输出流，那就在类定义中加入下列代码：

```
private static final SystemStream OUTPUT =
  new SystemStream("kafka","output");
```

这个SystemStream对象用来初始化一个OutgoingMessageEnvelope以及消息载荷和可选的键载荷。该对象随后被发送给MessageCollector，由它设法将输出消息放到指定的流中。

5.3.4.4 实现流任务

任务本身是通过StreamTask接口来定义的，所有的任务类都会实现该接口。这个接口只有一个处理方法，其输入参数为IncomingMessageEnvelope、MessageCollector和TaskCoordinator。IncomingMessageEnvelope是来自于task.inputs流的消息，这个流我们在上一节提到过。MessageCollector则用来发送事件到输出流，输出流是在类内部定义的。

例如，下列任务负责将wikipedia-raw主题的输入划分为单词，其划分方式与本章先前的Trident例子相同。这里，使用JSON SerDe来解析输入数据（JSON格式）并发送到wikipedia-words主题。

```java
public class WordSplitTask implements StreamTask {
  private static final SystemStream OUTPUT_STREAM
    = new SystemStream("kafka","wikipedia-words");

  public void process(IncomingMessageEnvelope envelope,
   MessageCollector collector,
   TaskCoordinator coordinator) throws Exception {
    try {
      @SuppressWarnings("unchecked")
      Map<String,Object> json =
        (Map<String,Object>)envelope.getMessage();
      String raw = (String)json.get("raw");
      raw = raw.substring(2,raw.indexOf("]]"));
      for(String word : raw.split("\\s+")) {
        HashMap<String,Object> val = new HashMap<String,Object>();
        val.put("word", word);

        collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,val));
      }
    } catch(Exception e) {
      System.err.println(e);
    }
  }
}
```

5.3.4.5 初始化任务和窗口

除了StreamTask接口，任务还可能实现InitableTask接口，这样就添加了一个init方法。该方法在创建任务对象时被调用，用来在运行时连接无法静态初始化的资源。

对于需要周期性地发送值的任务，例如计数操作，可以实现WindowableTask接口。该接口添加了一个window方法。Samza框架会周期性地调用该方法。

例如，上一节中消费wikipedia-words的输出的任务，由于需要维护各单词的计数，可能要使用所有这三个接口。这里，init接口其实并不是必需的，但完整性起见我们还是将其包含在内：

```java
public class WordCountTask implements StreamTask,
  InitableTask, WindowableTask {

  private static final SystemStream OUTPUT_STREAM =
    new SystemStream("kafka","wikipedia-counts");

  HashMap<String,Integer> counts = new HashMap<String,Integer>();
  HashSet<String> changed = new HashSet<String>();

  public void window(MessageCollector arg0, TaskCoordinator arg1)
    throws Exception {

    for(String word : changed) {
      HashMap<String,Object> val = new HashMap<String,Object>();
      val.put("word", word);
      val.put("count", counts.get(word));
      arg0.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,val));
    }
    changed.clear();
  }

   public void init(Config arg0, TaskContext arg1) throws Exception {
     counts.clear();
     changed.clear();
   }

   public void process(IncomingMessageEnvelope arg0,
     MessageCollector arg1,
     TaskCoordinator arg2) throws Exception {
     @SuppressWarnings("unchecked")
     Map<String,Object> json = (Map<String,Object>)arg0.getMessage();
     String word = (String) json.get("word");
     counts.put(word,
       (counts.containsKey(word) ? counts.get(word) : 0) + 1);
     changed.add(word);
   }
 }
```

该任务的属性文件指定了10秒的窗口周期：

```
# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=word-count

# YARN
yarn.package.path=
 file://${basedir}/target/
 ${project.artifactId}-${pom.version}-dist.tar.gz

# Task
task.class=wiley.streaming.samza.WordCountTask
task.inputs=kafka.wikipedia-words
task.window.ms=10000

# Serializers
serializers.registry.json.class=
org.apache.samza.serializers.JsonSerdeFactory

# Systems
systems.kafka.samza.factory=
org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
systems.kafka.producer.batch.num.messages=1
```

5.3.4.6 为YARN打包作业

要为YARN打包作业，必须创建一个用于发布的归档文件。该归档文件不仅包含实现作业的JAR文件，还应该包含所有的配置文件和项目的依赖。不仅如此，还需要有用YARN来启动作业的shell脚本。YARN会将该归档文件发布给将运行该作业的每个节点。

构造该归档文件最简单的方式就是使用Maven assembly插件。可以将该插件添加到pom.xml文件的plugins部分：

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.3</version>
  <configuration>
  <descriptors>
    <descriptor>src/main/assembly/src.xml</descriptor>
   </descriptors>
   </configuration>
   <executions>
   <execution>
     <id>make-assembly</id>
     <phase>package</phase>
     <goals>
       <goal>single</goal>
      </goals>
    </execution>
    </executions>
  </plugin>
```

该插件引用了一个assembly XML文件，该文件通常位于src/main/assembly.src目录。它定义了应该包含的文件和依赖。它首先定义要构建的程序集的类型，在这里是tar.gz文件：

```
<assembly
xmlns=
"http://maven.apache.org/plugins/maven-assembly-plugin/assembly
 /1.1.2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2
http://maven.apache.org/xsd/assembly-1.1.2.xsd"
>
  <id>dist</id>
  <formats>
    <format>tar.gz</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
```

接着包含了各种非依赖文件。其中最重要的是包含在工件中的作业配置文件：

```
<fileSets>
  <fileSet>
    <directory>${basedir}/..</directory>
    <includes>
      <include>README*</include>
      <include>LICENSE*</include>
      <include>NOTICE*</include>
    </includes>
  </fileSet>
</fileSets>
<files>
  <file>
    <source>${basedir}/src/main/config/word-split.properties</source>
    <outputDirectory>config</outputDirectory>
    <filtered>true</filtered>
  </file>
  <file>
    <source>${basedir}/src/main/config/word-count.properties</source>
    <outputDirectory>config</outputDirectory>
    <filtered>true</filtered>
  </file>

</files>
```

接下来的部分包含了来自samza-shell工件的shell脚本，并指定了在发布的tar.gz文件中应该包含的依赖：

```
<dependencySets>
  <dependencySet>
      <outputDirectory>bin</outputDirectory>
      <includes>
        <include>org.apache.samza:samza-shell:tgz:dist:*</include>
      </includes>
      <fileMode>0744</fileMode>
      <unpack>true</unpack>
    </dependencySet>
    <dependencySet>
```

```
      <outputDirectory>lib</outputDirectory>
      <includes>
        <include>org.apache.samza:samza-core_2.8.1</include>
        <include>org.apache.samza:samza-kafka_2.8.1</include>
        <include>org.apache.samza:samza-serializers_2.8.1</include>
        <include>org.apache.samza:samza-yarn_2.8.1</include>
        <include>org.slf4j:slf4j-log4j12</include>
<!--          <include>wiley:streaming-chapter-5</include> -->
        <include>org.apache.kafka:kafka_2.8.1</include>
      </includes>
      <useTransitiveFiltering>true</useTransitiveFiltering>
    </dependencySet>
  </dependencySets>
</assembly>
```

5.3.4.7　运行Samza作业

用于发布的归档文件构建完毕后，要将其复制到相应的位置。对于多节点Samza来说，应该复制到安装了合适的YARN客户端的机器上。

将归档文件解压到相应的目录，例如deploy目录：

```
$ mkdir deploy
$ cd deploy
$ tar xvfz ../target/streaming-chapter-5-1-dist.tar.gz
$ cd ..
```

然后使用归档文件中包含的run-job.sh脚本来将作业提交给YARN。例如，要提交本章的WordSplitTask和WordCountTask例子，运行下列命令：

```
$ ./deploy/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/config/word-split.properties
$  ./deploy/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/config/word-count.properties
```

最后检查wikipedia-counts主题的数据，以确保作业运行正常：

```
$ ./deploy/kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost:2181 --topic wikipedia-counts
```

5.4 总结

本章介绍了两个流处理框架：较为成熟的Storm和新近出现的Samza。这两个框架都比较新，要成为完整框架还有很长的道路要走。但是它们都可以并且已经用来建立有用的流应用。可以对其改造，使其满足几乎任何需求。随着它们不断成熟，它们必将越来越复杂。

流处理完成后，通常需要将流数据保存起来以备后用。下一章讨论这些流框架的存储选择问题，解决了流数据存储问题之后，前端应用就可以使用这些数据进行可视化或其他任务。

第6章　流数据的存储

构建流数据系统的一个主要原因是，要将系统不同部分之间的通信和访问解耦。这其中的一个关键就是数据存储和备份机制，这里所说的数据不仅包括原始数据，还包括已经由前面几章介绍的一个或多个处理环境处理过的数据。

数据处理只是一方面工作，数据还需要被交付给终端用户，因此还要将数据存储起来。可以将数据存储在处理系统中，这时可以使用Storm的分布式远程过程调用（DRPC）和bolt内部的内存等来存储。但是在生产环境中这是不现实的。首先，数据通常需要保留一段时间，这意味着对内存的要求过高。其次，对处理系统进行维护势必会中断所有的外部接口，尽管这两者本来没什么关系。最后，通常需要将处理结果保存到第三级存储（磁盘或“云”存储设备），这样更易于分析数据的长期趋势。

本章主要考虑如何存储处理后的数据。对于需要将数据交付给某种前端接口（通常是应用编程接口API或用户界面UI）的处理系统来说，在存储方面有许多选择。可能的选择有许多，但本章只讨论其中最常见的几个。我们刻意选择具有不同设计理念和设计约束的系统，以突出这些差异。这能帮助你在考虑特定应用的存储问题时做出更明智的选择。

对于长期的存储和分析任务，批处理系统往往要比流数据系统更靠谱。这多半因为流系统用相对昂贵的存储方案（如主存储器）来换取较低的随机访问延迟。批处理系统则恰恰相反，它们选择具有高容量和高随机访问延迟的存储（例如传统的旋转盘片存储器）。幸运的是，尽管批处理

系统的随机访问性能通常不如流系统，它们的线性读性能通常很出色，这使它们成为离线分析的最佳选择。

Hadoop已经成为这类批处理环境的黄金标准。本章会介绍如何搭建Hadoop以及怎样将Hadoop集成到流环境。

本章还将讨论以Hadoop为网关连接已有的业务智能基础架构的基础问题。讨论这个问题的原因是，尽管本书聚焦于实时流数据分析，但没有什么现代的分析系统是运行于真空之中的。如果想让一个系统被广泛采用，就必须与企业环境中的其他系统相集成。

6.1　一致性哈希

本章后面介绍的几个数据存储支持将数据分布到多个服务器中，这使得它们更容易随着数据规模的增大而扩展。此外，这样做往往还会提高数据的更新和查询性能。一致性哈希（consistent hashing）就是实现数据分布化的一项广受青睐技术。

多数数据存储有“键”的概念。在关系数据库中，它被称为主键。在键-值存储中则称为键。键的最重要的性质是，在数据存储中它的条目是唯一的（没有指定主键的关系数据库通常会创建任意的主键，这个主键对于用户是隐藏的）。

这些主键每一个都对应一个数值，这个数值则会被指定给特定的服务器。具体的指定机制多种多样，但最终的结果都是特定的主键总是被指定给相同的服务器，用于存储和查询。

这样做有助于将负载分布到多种服务器中，但也牺牲了可靠性。如果有任何服务器出现故障，那么存储在该服务器上的数据就会丢失。随着服务器数量的增加，一台服务器在给定时间内出现故障的概率也会增加。

为了提高系统的可靠性，需要对数据进行一致性哈希。在一致性哈希中，首先为服务器赋予一个特定的稳定顺序，称作环（ring）。当主键被修改时，初始服务器的选择跟以前一样，但是环中后面的k台服务器上的数据将会被修改（引入环的原因就在于，如果主服务器位于列表终端服务器的k台服务器范围之内，需要修改数据的服务器可以“回绕”到服务器列

表的开头）。

如果一台服务器发生故障，客户端就可以只更新和查询剩下的服务器。如果该服务器又回到服务器池，它可以将数据从其中一台其他服务器中复制回来，这时该服务器就拥有了最新的数据。

添加新服务器或从环中永久地删除服务器时，数据存储需要进行类似的过程。添加服务器时，新服务器会根据它在环中的位置复制数据。删除服务器时，则需要一个现有的服务器，用来保存从环中即将被删除的服务器中复制出来的数据。这两个操作都可以在后台进行。由于环中有其他服务器在提供数据查询服务，正在更新的服务器就是不可见的。来自客户端的数据更新可以直接应用到新服务器，不需要检查。

6.2 "NoSQL"存储系统

伴随着大数据应用的兴起，所谓的"NoSQL"存储系统在过去的几年间逐渐流行开来。本节宽泛地介绍了实现这类存储系统的不同方法，这些系统多半具有高性能的读写操作、牺牲事务性数据库的常见需求、查询很复杂等特点，并且往往兼而有之。

对于大数据应用，特别是实时流应用来说，上述折衷都被认为是可以接受的。特别是当使用本章后面讨论的Lambda架构时，可以将实时分析的数据存储需求放松到相当宽松的程度。

对于流应用有许多不同的存储选择，但本节我们只讨论其中三个。选择这三种存储系统，意在展示存储选择的可能范围。在生产系统中，很可能会同时使用其中的多个，因为它们各有优缺点。要说明的一点是，这些系统通常是关系数据库的补充，而不是取代者。

第一个存储系统是最简单的键-值存储。有许多不同的键-值存储，但Redis是其中最流行的。它的性能与Memcached系统类似，但提供了对高级数据结构的本地支持，这些数据结构对于许多流分析应用都很有用。它不支持分布式存储，主要支持高性能的单机应用。

第二个存储系统是文档存储系统MongoDB。MongoDB是无模式的存储系统。事实证明，对于需要维护大量配置文件或数据，并且这些文件和数据可以很自然地按序组织到文档中的应用程序，MongoDB是广受欢迎的存储系统选择。它支持主-从复制和分片（分区），还支持非常高的写性

能。其写性能几乎能达到Memcached系统的水平，只不过是以安全性为代价。在现代版的数据库中，可以在客户端层对这种权衡进行调整。

最后一个存储系统是分布式数据库系统Cassandra。通过借用Amazon的DynamoDB和Google的BigTable中的元素，Cassandra兼具了键-值存储和表式数据库的特点。它的早期版本因为查询语言难以使用而颇受诟病，这主要因为它实质上将内部数据结构开放给了用户。不过，最近的版本引入了与SQL语言非常相近的查询语言，比原来要好用得多，从而大大提升了自身的吸引力。

本节接下来的内容依次讨论这三种存储系统的用法。哪种系统是最好的？这个问题基本上是无解的。因为不存在适用所有应用的“最好”技术。这是否意味着可能要在一个环境中使用多种数据库技术？完全正确！这就是我们在前几章考虑构建灵活的数据流程系统的原因：数据需要在系统间流动和变换。

6.2.1 Redis

Redis是一个简单性与复杂性并存的键-值存储系统。说简单是因为，它没有花太多精力来解决键-值存储要经常解决的两个问题：工作集规模大于主存储器容量和分布式存储。Redis服务器只能为可以放入主存储器的数据提供服务。它有一定的数据复制功能，但不支持最终一致性等特性。对于分片和一致性哈希，Redis曾经考虑过其实现，但都无果而终。即使Redis支持这些功能，也都是由外部服务提供的。在许多方面，Redis更接近于Memcached这样的缓存系统，而不是数据库。

与缓存系统不同的是，Redis为具有原子更新能力的高层数据结构提供服务器端支持。除了基本的键-值功能，基本的数据结构还包括列表、集合、哈希表和有序集合。它还包括了键的过期功能以及发布-订阅机制，后者可以用作（例如实时流处理系统和前端之间）消息总线。举例来说，当键得到更新时，前端服务器会被告知该事件，这样用户界面就可以更新。对此，后面的几章会给出更多细节。

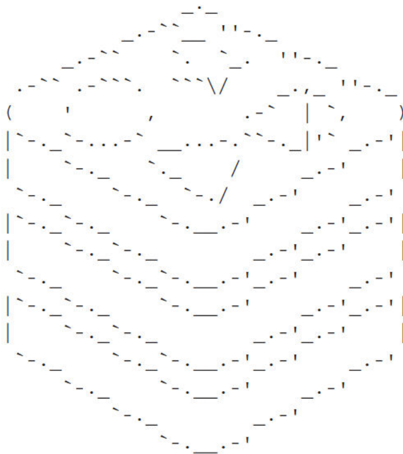
6.2.1.1　安装Redis

在多数操作系统中，都可以通过标准的软件包管理器获得Redis。如果Redis已经过时，或者无法通过软件包管理器来获得时，在类Unix的系统上构建它也是很简单的。下载相应的归档文件后，使用常用的make&&make install命令就可以构建合适版本的Redis。默认情况下构建的是64位Redis，但也可以构建32位版本。32位版本会便宜些，但会将Redis总容量限制为4GB。

注意　Amazon EC2服务的用户可以直接在Elasticache中找到Redis。这里的Redis通常不是最新的发布版，但不论在开发环境还是生产环境中，这样会让Redis新手快速上手。你只需要从下拉菜单中选择Redis就可以了（默认显示的是memcache）。

Redis自带有一系列很好的默认配置参数，但是当你使用Redis时，有一些选择需要仔细考虑。配置条目通常位于redis.conf文件中，它们会在启动时被传递给服务器。

```
$redis-server
```



```
Redis 2.6.16 64 bit

Running in stand alone mode
Port: 6379
PID: 23908

    http://redis.io
```

```
[23908] 09 Nov 14:16:16.249 # Server started, Redis version 2.6.16
[23908] 09 Nov 14:16:16.250 * The server is now ready to accept
  connections on port 6379
```

6.2.1.2 使用Redis

刚开始摸索Redis时，最简单的方式是使用redis-cli工具连接redis服务器。启动redis服务器的本地副本之后，打开redis客户端，其输出信息如下所示：

```
$ redis-cli
redis 127.0.0.1:6379> info
# Server
redis_version:2.6.16
redis_git_sha1:00000000

[...]

# Memory
used_memory:1082448
used_memory_human:1.03M
used_memory_rss:319488
used_memory_peak:1133520
used_memory_peak_human:1.08M
used_memory_lua:31744
mem_fragmentation_ratio:0.30
mem_allocator:libc

[...]

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:69.16
used_cpu_user:35.95


used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
> db0:keys=1,expires=0,avg_ttl=0
```

上述输出信息显示的是info命令的结果。为了节省空间，我们将有的输出省略掉了。这里面有一些值得注意的项目，这些项目在管理和监控Redis实例时会派上用场。

由于Redis是键-值存储系统，它的最基本的命令就是设置键的命令。Redis中的键通常被视为字符串值，即使它们表示的是整数或浮点数也是如此。要设置或获取键，应分别使用命令SET＜key＞＜value＞和GET＜key＞。如果要在一次调用中设置或获取多个键，可以分别使用命令MSET＜key1＞＜value1＞＜key2＞＜value2＞和

MGET＜key1＞＜key2＞，而不必多次调用。不管在什么情况下，如果键不存在，GET命令和它的变体都会返回NULL。如果只需要检查键是否存在，也可以使用EXIST命令。该命令通常与Redis的脚本功能一起使用，我们会在后面讨论脚本功能。

Redis的下一个复杂数据结构是哈希表。这种形式的键允许存储子键，这一点与Java中的Map集合或JavaScript等语言的字典类型很相似。这时你就不能再使用简单的GET和SET命令，而应该使用HSET＜key＞＜subkey＞＜value＞和HGET＜key＞＜subkey＞来访问哈希表的元素，使用HMSET和HMGET从哈希表更新或获取多个子键。要获取所有的子键及其值，应使用HGETALL，其中HKEY和HVALUES分别获取所有的子键名和子键值。

常规的键和哈希表都支持原子计数器。对于常规的键来说，INCR将键的值加1，INCRBY命令则将键的值增加指定的值。这两种情况下，键中包含的字符串都被解释为64位有符号整数，以进行算术运算。计数器的减法是通过DECR和DECRBY命令来实现的。INCRBYFLOAT命令与INCRBY类似，唯一的不同是它将值中包含的字符串解释为双精度浮点数值。

哈希表的原子计数器作用于子键。哈希表并没有实现全套的计数器命令，只实现了INCRBY和INCRBYFLOAT的变体，分别称作HINCRBY和HINCRBYFLOAT。

提示　初看起来，既然有了键-值存储，带有子键的哈希似乎没有存在的必要。但是，许多情况下使用哈希要比只使用键更好。首先，如果有几个不同的度量（metric）要存储，使用哈希存储可能要比单独使用键实

际上更高效。这是因为在Redis的哈希表实现中对小的哈希使用ziplist结构来提高存储性能。其次，如果这些度量都是相关的，可以使用HGETALL命令在一条命令中获取所有度量。哈希结构自身的性质使得所有的键仍然是独立更新的，因此这样做没什么不利之处。最后，Redis数据库中的数据每过一段时间就要过期，这种情况很常见。通常这是为了控制数据库的内存占用量。如果使用哈希表，那就意味着只需要让单个键过期，这有助于维护一致性。

Redis的列表（list）数据结构常被用来实现各种类队列结构。你可以通过列表数据类型的命令集看到一点，这些命令的主要作用是向列表中压入元素或从表中弹出元素。基本的命令是LPUSH（RPUSH）和LPOP（RPOP）。Redis列表是双向的，因此左右插入的性能是相同的。使用同“一边”的命令（例如LPUSH和LPOP），实现的就是栈语义，即后进先出（LIFO）语义。使用相反一边的命令（例如LPUSH和RPOP），则会实现队列语义，即先进先出（FIFO）语义。

实现一个可靠队列

通过利用RPOPLPUSH（或其阻塞版本）和LREM命令，Redis提供了一个实现可靠队列的非常简单的机制。根据RPOPLPUSH这个名字你应该可以猜出来，在一个原子操作内，该命令像RPOP命令那样弹出列表末尾的元素，然后像LPUSH命令那样将其压入第二个列表的开头。与分开使用RPOP和LPUSH命令不同，RPOPLPUSH能够确保数据不会在客户端间传输的过程中丢失，因为要处理的数据项会保留在保存当前处理元素的“传输中”列表中。

该命令会返回在两个列表中移动以进行处理的元素。当处理完成时，会对“处理中”列表执行LREM命令，从“处理中”列表中删除该元素。下面这个简单的例子展示了实现队列的基本过程：

```
redis 127.0.0.1:6379> LPUSH queue item1 item2 item3 item4 item5
(integer) 5
redis 127.0.0.1:6379> RPOPLPUSH queue processing
"item1"
redis 127.0.0.1:6379> LRANGE queue 0 100
1) "item5"
2) "item4"
3) "item3"
4) "item2"
redis 127.0.0.1:6379> LRANGE processing 0 100
1) "item1"
redis 127.0.0.1:6379> LREM processing 0 item1
(integer) 1
redis 127.0.0.1:6379> LRANGE processing 0 100
(empty list or set)
redis 127.0.0.1:6379>
```

“实现一个可靠队列”这个例子演示了将多个项压入queue列表，然后将queue列表中的一个元素移到“处理中”列表，最后将该元素从“处理中”列表中删除。

当使用客户端实现时，更常见的做法是使用BRPOPLPUSH命令。这是常规命令的阻塞版本，它会一直等待，直到元素被压入列表中。阻塞版命令避免了轮询客户端。该命令支持超时特性，以防客户端需要执行其他函数的情况。

通常会实现另外一个客户端来处理被放置到“处理中”列表中但却未被删除的元素。这个负责善后工作的客户端可以周期性地扫描列表，如果看到一个元素两次，就将其压回queue列表重新处理。无需从“处理中”队列中删除原始项，如果LREM成功执行，它会删除元素的所有副本，“处理

中”队列通常会始终保持较小规模，因此该队列大小不成问题。

Redis实现的另外两个数据类型是集合（Set）和有序集合（Sorted Set）。与Java的Set集合类似，这些数据结构负责维护不同值的列表。集合和有序集合的多数操作的处理时间都可以用集合大小表示为O（n）。

基本的集合是非常简单的概念，它对于在实时流中跟踪唯一的事件或对象很有用。当集合大小变得十分巨大时，基本的集合就无能为力了，这主要因为它需要的空间与集合中不同元素的数量呈线性关系。向集合中加入元素是通过SADD命令完成的，该命令以一个KEY和要加入的任意数量的元素为输入。其返回值为新加入集合的元素数量。要将元素从集合中删除，应使用SREM命令。该命令的参数与SADD命令相同。SPOP命令与SREM类似，但返回的是集合中的一个随机元素并将其删除。SMOVE命令将SADD命令和SREM命令合并为一个原子操作，从而在两个不同的集合间移动元素。

集合成员查询是用SISMEMBER命令完成的。可惜的是，该命令只允许将一个元素作为参数。要实现成员查询的“any”或“all”变体，就必须执行多条SISMEMBER命令，这时就要用到本节随后介绍的Redis脚本功能。

Redis还支持数据库中不同集合对象间的集合运算。SUNION、SINTER和SDIFF命令分别返回包含两个集合的并集、交集或差的批量结果。它还可以通过向这些命令加入STORE修饰符将结果存储到其他集合。例如，要将两个集合的并集存储到第三个集合，可以使用SUNIONSTORE命令。还可以使用SCARD命令来获取任何集合的大小。

有序集合向集合的每个元素加上一个分值，用这个分值来定义集合中元素的排序。这在需要维护排行榜以及其他类似的结构时很有用。一般来说，更新有序集合的命令与更新常规集合的命令是基本相同的，只是把命令的前缀从S换成Z。此外，还有一个额外的参数，即“分值”。对于ZADD命令，该分值处于要加入的每个元素的前面。如果元素已经存在，就将已有的分值更改为新的分值。

有序集合是通过ZUNIONSTORE和ZINTERSTORE命令来支持并集和交集操作的。这些操作与非有序集合的对应操作略有不同，因为它们需要处理两个集合的共有元素具有不同分值的情况。这两个操作都有一个可选的WEIGHTS参数，用来定义用在每个集合分值上的倍增因子。例如，WEIGHTS 23会将第一个集合的分值乘上2，第二个集合的分值乘上3。默认情况下，对于同时出现在两个集合中的元素，会将这两个分值加到一起。不过，这可以通过可选的AGGREGATE参数修改。该参数可能的取值为SUM（默认）、MIN或MAX。当选择了SUM时，两个或更多集合的共有元素的分值会在倍增之后加在一起。当使用MIN时，会使用最小的权重值。如果是MAX，则使用最大的权重值。

有序集合还引入一些新的命令，这些命令要用到与每个元素关联的分值。ZINCRBY命令增加集合中某个元素的分值。如果该元素不存在，就将其分值设为0.0，并将该元素插入有序集合。

ZCARD命令与SCARD类似，它返回整个集合的基数（cardinality）。此外，ZCOUNT命令返回分值处于两个分值之间的元素数量。例如，下列命令

```
ZCOUNT myzset 1 3
```

返回分值在1和3之间的元素数量，包括分值等于1和3的元素（即 $1 \le x \le 3$）。如果不想计入分值为1和3的元素（即 $1 < x < 3$），那就在分值之前加上一个前括号“（”：

```
ZCOUNT myzset (1 (3.
```

要获取某个范围内的元素，而不是元素的数量，应该使用ZRANGEBYSCORE和ZREVRANGEBYSCORE命令。这两个命令的参数与ZCOUNT相同，但切记ZREVRANGEBYSCORE要求第一个参数大于第二个参数。例如，

```
ZREVRANGEBYSCORE myzset 1 3
```

不会返回任何元素，而

```
ZREVRANGEBYSCORE myzset 3 1
```

才会返回我们想要的结果。这两个命令都有一个可选的参数WITHSCORES，如果使用了该参数，命令就会同时返回元素和其分值。这两个命令还支持LIMIT参数，该参数的取值为一个偏移量和用于实现分页接口的计数值。使用该偏移量需要遍历有序集合，因此对于非常大的偏移量来说，命令的执行可能会很慢。

对于排行榜这类应用，ZRANGE和ZREVRANGE会返回位置start和位置stop之间的元素。起始位置都是从0开始的，因此要获取分值最高的前10个元素，使用

```
ZREVRANGE myzset 9 0
```

就会返回所需要的值。与许多编程语言类似，以负位置使用ZRANGE命令也有相同的效果。位置—1是集合最末尾的元素，依此类推。

6.2.1.3　脚本

Redis 2.6引入了内置的Lua脚本引擎，以最终取代它已有的事务功能。可以通过EVAL命令来使用该功能，命令的形式如下：

```
EVAL <script> <num keys> <key 1> ... <key n> <arg 1> ... <arg n>
```

脚本有时会很长，为此Redis还提供了EVALSHA命令，该命令用固定大小的SHA1哈希值来替代实际的脚本。如果redis服务器无法根据SHA1哈希值找到到对应的脚本，它就会返回特殊的错误，该错误会让EVALSHA退化为常规的EVAL命令。多数Redis客户端会在执行EVAL命令时隐式地使用EVALSHA，因此用户通常没有必要考虑该命令。

不论客户端连接的情况怎样，所有的脚本都是在单个Lua解释器上运行的，因此每个脚本都可以被视为原子操作。

跟踪最近发生的事件

Redis引擎中直接包含了脚本语言，这使得一些原来很难实现的任务变得易如反掌。“跟踪最近发生的事件”就属于这种任务。该任务常被用于归因建模（attribution modeling），即某些目标事件与最近发生的事件相关。

在多数键-值存储中，要实现这项任务，通常假设事件的处理是按照“足够好的”时间顺序进行的，这样与给定用户相关的任意两个事件都不

大可能“乱序”。如果是这种情况，使用简单的SET操作就足够了。

但是在许多情况下，这个顺序假定并不成立，或者需要更加复杂的安排。例如，可能某些类型的事件要比其他事件更重要，因此假设一个事件是后发生的，如果它是次要事件，那就不应成为“最近”的事件。在常规的键-值系统中，这需要与服务器的多轮交互。首先要获取当前值，这样才可以应用更新逻辑。如果更新成功，就在第二轮更新事件。

使用Lua脚本，就可以完全在服务器端完成这种更新过程。假设事件是JSON对象，包含一个ts字段，该字段表示事件的时间戳，更新脚本应如下所示：

```
if redis.call("EXISTS",KEYS[1]) == 1 then
  local current  = tonumber(
   cjson.decode(
    redis.call("GET",KEYS[1])
   )["ts"]
  )
  local new      = tonumber(cjson.decode(ARGV[1])["ts"])
  if(new >= current) then
    redis.call("SET",KEYS[1],ARGV[1])
    return 1
  else
    return 0
  end
else
  redis.call("SET",KEYS[1],ARGV[1])
  return 1
end
```

如果要试用一下该脚本，那就将上述脚本保存为timestamps.lua，或者使用本书源代码中包含的脚本。启动Redis服务器之后，设置几个带有时间戳的不同事件：

```
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":0}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":2}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":10}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":5}'
(integer) 0
$ redis-cli GET user
"{\"ts\":10}"
```

尽管时间戳本来是乱序的，但最近的时间戳仍然保持了顺序。如果要对JSON对象进行更复杂的测试，可以将相关代码放在上述代码的第4行后面。这样做也具有原子操作的优点，因为向用户的更新只是通过脚本完成的，而脚本始终是在同一个Lua解释器中执行的。

6.2.1.4　对发布/订阅的支持

除了作为键-值存储，Redis还可以用作简单的消息总线。这在流应用中特别有用，因为它允许Storm这样的处理系统向前端通告重要事件。我们在第7章中给出了这样的例子，该例子使用Redis的消息总线功能实现基本的实时仪表板。

Redis中的事件是简单的文本，用PUBLISH命令可以将事件发布到通道（channel）中：

```
PUBLISH achannel "some message"
```

这些“PUBLISH”命令会被传递给已经用SUBSCRIBE命令订阅了一个或多个通道的客户端。为了方便起见，这些命令会像Redis中的其他任何命令一样被复制，因此它们会被传递给已经向主机器（master）的从机器（slave）订阅的客户端。但是，所有的发布命令都必须在主机器上发出。

6.2.1.5　复制

Redis支持基本的主-从复制。通常来说该功能很好用，但在2.6版本系列中，主-从之间一旦失去同步就需要完全复制。对于大规模系统，这可能需要很长时间，如果复制失败，就必须重新开始。当在数据中心之间尝试复制Redis数据库时，麻烦会更大。

目前的Redis beta版，将会发布为2.8系列，它支持部分再同步。这就降低了不可靠网络链路和使用Redis带来的，由于从机器无法在合理的时间内完成初始同步而导致集群无法使用等诸多问题。

设置Redis的主-从复制环境很简单。主机器通常并不需要做任何改变，从机器只需要向其配置文件加入slaveof指令：

```
slaveof <master ip address> <port>
```

你甚至可以在Redis命令行客户端上使用这条指令，使用CONFIG SET命令就不需要重启服务器应用。不过，你要仔细确保这条指令已经加入服务器配置中，否则如果服务器由于某种原因重启，对配置所做的修改就会丢失。由于Redis的主服务器不需要做任何专门的配置，因此实际上可以将Redis服务器的从机器改为其他机器。这种情况的应用场景有限，不应在生产环境中使用这种方法，但是如果你维护一个本地的Redis开发服务器，并且需要访问生产环境的数据，这种做法有时比较有用。

6.2.1.6　集群数据库/共享数据库

目前Redis并不支持像许多其他“NoSQL”数据库那样的本地共享解决方案。曾经有人研发过一个称为Redis Cluster的共享版Redis，但这个版本

从来没有以产品的形式发布过。

现如今共享Redis数据库的唯一现实的选择就是Twitter的twemproxy工具（又名nutcracker）。它是Redis和Memcached的代理服务器，为这两个键-值存储系统实现了共享特性。使用twemproxy可以将服务器合并到服务器池中，并用一致性哈希将写操作在服务器池中的成员中分布化。它支持多台服务器间的数据冗余和分区。其缺点是不支持Redis的某些操作，因此如果使用了复杂脚本或发布/订阅机制，又没有额外的架构方面的支持（例如保持一个共享和非共享的服务器池），就无法使用twemproxy。

6.2.2　MongoDB

MongoDB与Redis不同，它是无模式的文档存储系统。它是由10gen公司开发的，该公司提供MongoDB安装的企业级销售和技术支持。根据企业自身的需求，可以以软件即服务（Software-as-a-Service）或企业私有模式（On-Premise model）来提供MongoDB。MongoDB还有社区版，该版本不具备企业版支持的许多安全特性。这一节我们讨论的就是这个版本。

6.2.2.1　MongoDB模型

MongoDB中的文档是用JSON对象表示的无模式数据结构，他们类似于传统数据库系统中的记录或键-值存储系统中的值。

这些文档中每个文档都有一个唯一的标识符，可以用来直接寻址。标

识符的设计初衷却是希望通过将文档分组成集合（collection）以批量处理。这些集合大体上类似于关系数据库系统中的表。我们希望一个集合中包含的文档对象具有基本相同的结构。

可以通过基于JSON的查询语言对这些集合进行查询。为了提高性能，每个文档的元素都会被二级索引系统索引。MongoDB提供了多种有趣的索引选择，这使得它成为某些特定应用的广受欢迎的数据存储系统。特别对于要处理物理空间的应用，MongoDB更是不二选择，这是因为它具备对地理信息的索引和查询的现成支持。

6.2.2.2 安装MongoDB

要准备安装MongoDB，可以登录10gen MongoDB网站。在http://mongodb.org/downloads 上有针对许多平台的安装包。与Redis和许多其他NoSQL数据库不同，MongoDB将Windows系统也作为“一等公民”来支持。对于在linux系统上的部署，它提供针对Ubuntu和Debian的安装包，这样在EC2这样的平台上进行部署也会很容易。

MongoDB服务器的可执行文件的名称是mogod，如果无参数启动它，它就会自行查找/data/db目录。你可以通过在命令行中传递--dbpath参数来覆盖查找目录：

```
$ mkdir db
$ ./mongod --dbpath ./db
[initandlisten] MongoDB starting : pid=6722 port=27017 dbpath=./db
[initandlisten]
[initandlisten] ** WARNING: soft rlimits too low. Number of files is
  256, should be at least 1000
[initandlisten] db version v2.4.8
[initandlisten] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
[initandlisten] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
[initandlisten] allocator: system
[initandlisten] options: { dbpath: "./db" }
[initandlisten] journal dir=./db/journal
[initandlisten] recover : no journal files present, no recovery needed
[FileAllocator] allocating new datafile ./db/local.ns, filling with
  zeroes...
[FileAllocator] creating directory ./db/_tmp
[FileAllocator] done allocating datafile ./db/local.ns, size: 16MB, took
[FileAllocator] allocating new datafile ./db/local.0, filling with
[FileAllocator] done allocating datafile ./db/local.0, size: 64MB, took
  0.327 secs
[initandlisten] command local.$cmd command: { create: "startup_log",
  size: 10485760, capped: true }
  ntoreturn:1 keyUpdates:0  reslen:37 470ms
[websvr] admin web console waiting for connections on port 28017
[initandlisten] waiting for connections on port 27017
```

服务器启动时显示的信息多数只是通告性消息，但有些还是值得注意的。首先，关于soft rlimits的警告很重要。多数操作系统（包括Linux）默认假设应用离“消费”系统要比“服务器”系统（特别是数据库服务器）近得多，因此将内存和文件打开数量方面的许多限制都设置得过低，导致在生产环境中无法使用。即使对外宣称为“服务器”版本的操作系统也有此类问题。对于开发环境来说，可能没有必要提高这种限制，但生产环境服务器应该将这个限制设置得相当高才行，因为大型数据库随时都可能用到成千上万的资源。

另外一个要注意的是，MongoDB在首次启动时预先分配了一些文件。这些文件都被简单地用0填充，因为它们将由MongoDB通过内存映射来填入数据。如果大体知晓磁盘上的数据规模，那么就可以在启动MongoDB之前将这些空文件创建好，从而加快初始化的性能。否则，每当要创建新文

件时，MongoDB都要暂停，当服务器加入已有的服务器集群和复制已有数据时，这种情况会很频繁地出现。在类UNIX系统中，这很容易完成，可以使用dd命令来创建空文件。例如，下列MongoDB只包含名为local的初始系统数据库：

```
$ ls -lh
total 163840
drwxr-xr-x  2 bellis  staff    68B Dec  6 21:29 journal
-rw-------  1 bellis  staff    64M Dec  2 20:07 local.0
-rw-------  1 bellis  staff    16M Dec  2 20:07 local.ns
```

要加入一个名为big的新数据库，使用dd命令创建每个大小均为2GB的文件。该数据库需要大约4GB的空间，因此创建两个文件：

```
$ dd if=/dev/zero of=./big.0 bs=1048576 count=2048
2048+0 records in
2048+0 records out
2147483648 bytes transferred in 8.599453 secs (249723278 bytes/sec)
$ dd if=/dev/zero of=./big.1 bs=1048576 count=2048
2048+0 records in
2048+0 records out
2147483648 bytes transferred in 13.700213 secs (156748195 bytes/sec)
$ ls -lh
total 8552448
-rw-r--r--  1 bellis  staff   2.0G Mar  7 11:42 big.0
-rw-r--r--  1 bellis  staff   2.0G Mar  7 11:42 big.1
drwxr-xr-x  2 bellis  staff    68B Dec  6 21:29 journal
-rw-------  1 bellis  staff    64M Dec  2 20:07 local.0
-rw-------  1 bellis  staff    16M Dec  2 20:07 local.ns
```

6.2.2.3 使用MongoDB数据库和集合

MongoDB数据库是一个名字空间，在其中维护了一组集合（collection）。默认情况下，MongoDB只有一个名为local的数据库，其中包含了系统级的集合。尽管可以在local数据库中创建集合，但这样做绝对是个糟糕的主意，因为这样的集合不会被包含在复制或共享中。实际上，local数据库包含的是用来管理复制和共享的专门集合。

MongoDB的磁盘表示使用数据库名作为文件名，后面加上分段号。文件是以lazy模式创建的，因此直到数据库的第一个集合创建成功，数据库才会“存在”。举例来说，假设你尝试用mongo客户端切换到新安装的名为mine的数据库：

```
> use mine;
switched to db mine
> show databases;
local   0.078125GB
test   (empty)
```

注意，这时列出来的是local数据库，而不是mine数据库。但是，执行createCollection命令之后，mine数据库就会出现：

```
> db.createCollection("first");
{ "ok" : 1 }
> show dbs;
local   0.078125GB
mine   0.203125GB
test   (empty)
```

检查启动时由dbpath指定的目录，你可以看到第一个分段文件已经存在于磁盘中了：

```
$ ls
_tmp     journal     local.0     local.ns   mine.0     mine.1     mine.ns
mongod.lock
```

使用db.createCollection命令创建的集合是MongoDB中十分松散的概念。它们大体上类似于表，但因为MongoDB是无模式的，因此集合中的任何文档都不需要与其他文档相似。集合内的每个文档都表示为JSON对象，所有文档都有一个主键字段_id，这个字段的值通常是自动生成的ObjectId对象。不过，_id字段可以是除Array之外的任何类型的对象，只要它在集合中是唯一的就可以。对于流分析而言，使用这个字段作为某种

复合主键往往很有用，因为不用首先查询集合来找到相应的_id值（或者使用更复杂的更新查询），就可以更新特定的文档。

文档中的字段不得以$打头。$符号是为MongoDB文档中的特别对象专门保留的，其中最常见的就是DBRef对象，它支持MongoDB文档间的相互链接。具体来说，这些对象是包含$ref、$id和$db（可选）字段的JSON对象。$id字段用来确定目标文档的_id字段值；$ref字段用来确定目标集合；$db字段用来确定目标数据库（如果该数据库不是当前数据库）。例如，对第一个集合中某个文档的引用如下所示：

```
{$ref:"first",$id:ObjectId("5126bc054aed4daf9e2ab772"),$db:"mine"}
```

尽管集合中的文档可以是高度嵌套的，但一个文档的大小最多为16MB。从MongoDB开发人员的角度看，这个限制有点武断，但在数据建模过程中应该警惕过大文档的这一意义上，这个限制无可厚非。

6.2.2.4 固定集合

MongoDB支持将新集合创建为“固定集合”（capped collection）。这类集合的大小是固定的，目标是支持高吞吐量的操作。例如，MongoDB使用固定集合来为自己的复制策略实现操作日志。

用固定集合实现FIFO（先进先出）队列有几个限制。首先，固定集合不可以共享。其中一个原因是，这会让FIFO队列难以维护。其次，尽管写到固定集合的文档可以改变，但它们的大小不能增加。例如，将字符串类型的字段从“foo”变为“bar”没什么问题，但从“foo”变为“foobar”就不行了。最后，不能将文档从固定集合中删除。可以因为文档“老化”将其删

除，但无法直接删除指定的文档。

将集合转换为固定集合

可以将常规的集合转换为固定集合。该命令并不属于标准操作集，但可以通过runCommand方法来访问它。例如，将集合first转换为存储1MB数据的固定集合：

```
> db.runCommand({"convertToCapped": "first", size: 1024*1024});
```

运行这个命令的时候要小心，因为它带有一个全局的写锁。除非转换完成，向数据库的所有写操作都会阻塞。千万不要轮流在一个服务器上尝试这个命令。

可以用传统的方式查询固定集合，但更常见的方式是使用“tailable cursor”来以数据插入的顺序读取数据。这基本上等价于使用Unix的tail–f命令。

固定集合最常援引的用例是用作数据传输机制。在本书中使用的传输架构中，它可以替代数据传输层中的Kafka或Flume。不过，作为数据传输工具的固定集合有一个比较大的缺陷，因为固定的是集合规模，而不是时间。正如第5章所讨论的，这种方式有一些很难监控的故障模式。因此，不建议在数据传输中使用MongoDB的固定集合。

6.2.2.5　基本索引

集合的主要作用是维护一组文档共同的索引集。这里的假设是索引代表了MongoDB集合中文档的共同之处。

MongoDB中的索引是用db.collection.createIndex函数和一个JSON对象来定义的，这个JSON对象定义了索引中包含的每个字段的排序方向：

```
> db.first.ensureIndex({foo:1,bar:-1});
> db.first.stats()
{
  "ns" : "mine.first",
  "count" : 0,
  "size" : 0,
  "storageSize" : 8192,
  "numExtents" : 1,
  "nindexes" : 2,
  "lastExtentSize" : 8192,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 0,
  "totalIndexSize" : 16352,
  "indexSizes" : {
    "_id_" : 8176,
    "foo_1_bar_-1" : 8176
  },
  "ok" : 1
}
```

该命令在先前建立的first集合上建立一个索引，first集合有两个字段：foo和bar。Foo字段的内容是自然的升序（对数值是按数字排序，对字符串是按字典序），bar字段则是降序。使用点表示法，可以索引文档中的嵌套字段。例如，下列代码在文档中addr对象的zip子字段上定义了索引：

```
> db.first.ensureIndex({"addr.zip":1});
> db.first.getIndexes();
[
  {
    "v" : 1,
    "key" : {
      "addr.zip" : 1
    },
    "ns" : "mine.first",
    "name" : "addr.zip_1"
  }
]
```

当所索引的字段包含Array对象而不是标量对象时，MongoDB会自动建立一个多键索引。数组的每个元素都会被分别加入到索引中，因此包含四个元素的数组会分四次加入索引。包含对象的数组也采用这种方式。在前面的例子中，如果addr字段包含地址对象数组，每个地址对象的zip字段都会被加入到索引中。

6.2.2.6　地理空间索引

除了基本的标量字段索引，MongoDB还支持多种专门的索引类型，这使得它成为某些应用的首选。第一种索引就是地理空间索引，这种索引作用于坐标对或GeoJSON对象。坐标对是MongoDB特有的一种旧有格式，可以表示为包含两个元素的数组或包含lng和lat字段的对象。下列两种表示是等价的：

```
{lng: 10, lat: 20}
[10,20]
```

GeoJSON对象是较新的开源格式，用来描述地理空间特征。如果用GeoJSON来表示坐标对，其格式应该是：

```
{type:"Point",coordinates:[10,20]}
```

除了Point类型，MongoDB索引还支持Line和Polygon类型。

向索引添加字段时，MongoDB可以不指定数字，而是指定2dsphere、2d或haystack来索引这些坐标。对多数应用程序来说，2dsphere可能是索引的最佳选择。不过，如果要索引的区域是较小的地理空间，就可以使用haystack索引以提高性能。

6.2.2.7　全文本索引

在建立索引时，如果将字段的索引类型设置为text而不是数字，那就启用了MongoDB的全文本索引。全文本索引功能目前还处于测试阶段，因此如果要启用该功能，在启动服务器时必须在命令行中加上--setParameter textSearchEnabled=true。

这类索引在许多方面都类似于数组字段所用的多键索引。两者的主要区别在于我们希望全文本索引的字段是字符串，该字符串会被标记化和切分词干以形成输入索引的词。MongoDB还维护特定语言的停用词列表，这些停用词会在索引过程被忽略掉。

对一系列搜索项的查询是通过计分系统计算的。对不同的词，MongoDB为它所支持的每种语言都维护一个分值，但可以在建立索引的时候重写该分值。在后面一节中，我们会给出对应这个选项的可选参数。

全文本索引占用的处理资源不可小觑，因此应小心使用。如果向全文本索引的集合插入文档，也会伴随有与标记化和切分词干过程相关的处理代价，因此对于高吞吐率的集合，我们不建议使用全文本索引。

6.2.2.8　可选的其他索引

建立索引时，会自动赋予它一个索引名。该名称加上集合名最长为125字符。当建立复杂索引时，可能会超出这个限制。为了避免发生这种情况，也为了美观起见，可以在建立索引时使用选项对象（option object）来命名索引。选项对象还支持各种其他可选参数，如表6-1所示。

表6-1 建立MongoDB索引的可选参数

名　　称	类　　型	描　　述
background	true/false	使索引可以在后台建立。如果为 false，建立索引时数据库不会有响应。如果集合很大，这可能意味着几分钟的停机时间
unique	true/false	索引中的所有值必须是唯一的。对 id 字段自动创建的索引会设置该参数
name	string	覆盖自动生成的索引名
dropDups	true/false	删除与所发现的第一个文档具有相同索引值的文档，从而建立唯一（unique）的索引。要小心使用
sparse	true/false	只包括含有索引指定字段的文档。如果许多文档都不包含某个特定的字段，创建的索引会比较小
expireAfterSeconds	integer	根据该字段所用的时间戳决定文档的生存时间。设置这个选项会让索引成为 TTL 索引。它不能用于组合索引，并且该字段必须是日期类型
v	version	不同版本的 MongoDB 使用不同的索引格式和模式。可以用这个参数来选择索引模式。通常没有必要使用这个参数
weights	JSON	对于全文本索引，设置这个参数可以赋予不同单词不同的权重。这可以用于在查询期间计算搜索分值
default_language	string	定义全文本索引引擎所用的语言。该参数定义了停用词表和词干切分过程以及单词的默认分值
language_override	string	在文档层次定义可以用来覆盖全文本索引所用语言的字段。默认是字段 language

来源：http://docs.mongodb.org/manual/reference/method/db.collection.ensureIndex/#db.collection.ensureIndex

6.2.2.9 插入和更新

MongoDB支持对其集合进行常见的插入和更新操作。Insert命令可以插入单个元素。如果传入的输入参数是数组，也可以用一条命令插入多个文档。

```
> db.first.insert({metric:"test",n:1});
> db.first.insert([{metric:"test2",n:1},{metric:"test3",n:1}]);
> db.first.find();
{ "_id" : ObjectId("529c1bb69c88a09b200d9cac"), "metric" : "test",
  "n" : 1 }
{ "_id" : ObjectId("529c1bca9c88a09b200d9cad"), "metric" : "test2",
  "n" : 1 }
{ "_id" : ObjectId("529c1bca9c88a09b200d9cae"), "metric" : "test3",
  "n" : 1 }
```

如果没有指定键，insert命令会自动创建_id键。为了易于更新，多数流应用会显式地指定_id：

```
> db.first.insert({_id:"test:201312010000"
  ,ts:new Date(2013,11,01,00,00),metric:"test",n:1});
> db.first.find();
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"), "metric" : "test", "n" : 1 }
```

如果尝试插入具有重复键的文档，插入会失败，并报错：

```
> db.first.insert({_id:"test:201312010000",
  ts:new Date(2013,11,01,00,00),metric:"test",n:1});
  E11000 duplicate key error index: mine.first.$_id_
  dup key: { : "test:201312010000" }
```

文档的更新是通过update命令完成的。该命令的输入参数通常有两个，一个是对_id字段的匹配条件，另一个是要替换的文档。默认情况下，该命令会完全替换掉所匹配的文档，因此将复制完整的文档。例如，下面的结果可能并不是该更新真正需要的结果：

```
> db.first.update({_id:"test:201312010000"},{n:2});
> db.first.find()
{ "_id" : "test:201312010000", "n" : 2 }
```

请注意我们很可能只需要更新字段n，而事实上整个文档都被替换掉了。要做到只更新字段n，使用特殊字段$set来定义要更新的字段：

```
bb.first.drop();
> db.first.insert({_id:"test:201312010000",
  ts:new Date(2013,11,01,00,00),metric:"test",n:1});
> db.first.update({_id:"test:201312010000"},{"$set":{n:2}});
> db.first.find()
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"), "metric" : "test", "n" : 2 }
```

除了$set，更新命令也可以用$inc来增加数字字段：

```
> db.first.update({_id:"test:201312010000"},{"$inc":{n:2}});
> db.first.find()
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"),
  "metric" : "test", "n" : 4 }
```

一个MongoDB度量集合

在MongoDB中创建一个简单的度量集合是很容易的事情，用更新命令的upsert特性就可以做到。首先创建一个度量集合，并定义一个带有生存时间的索引以使数据在7天后过期：

```
> db.createCollection("metrics");
{ "ok" : 1 }
> db.metrics.ensureIndex({ts:1},{expireAfterSeconds:86400*7});
> db.metrics.getIndexes();
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "mine.metrics",
    "name" : "_id_"
  },
  {
    "v" : 1,
    "key" : {
      "ts" : 1
    },
    "ns" : "mine.metrics",
    "name" : "ts_1",
    "expireAfterSeconds" : 604800
  }
]
```

理想的情况是使用_id字段作为日期字段，这样每个日期都是唯一的，但可惜的是这样做行不通。要进行聚集的时间戳必须保存在两个键中。从长远来看这基本上是可取的，因为度量可能会被细分为更小的粒度。在这个例子中，将customer_id加入到_id中来支持对每个客户的度量进行聚集。

这个聚集本身是通过更新命令完成的：

```
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.visitors":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.clicks":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.views":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.visitors":1}},{upsert:true});
> db.metrics.find();
{ "_id" : "1:201312010000", "customer_id" : 1,
  "metrics" : {
    "clicks" : 1,
  "views" : 1,
    "visitors" : 2
  },
  "ts" : ISODate("2013-12-01T08:00:00Z")
}
```

注意upsert选项被设置为了true。这样，当匹配的文档不存在时，会创建该文档。正因如此，每次调用都要设置customer_id和ts字段，只有这样，才能在必须创建文档时确保它们已经被设置为正确的值。

6.2.2.10　查询和聚集计算

更新命令使用简单查询来确定要修改的文档。总的来说，MongoDB有一个非常丰富的查询语言，不过由于该语言与文档一样都是用JSON表达的，因而往往比较繁琐。查询是用find命令来进行的，查询的性能严重依赖对集合的索引。如果无法使用索引，那么就要扫描集合中所有的文档。

上一节中使用的查询是简单的匹配查询，它们等价于在SQL查询中的WHERE子句中使用“＝”号。如果要查询的字段是数组，该等值查询会返

回某些文档，这些文档所包含的数组中的任何元素都与查询值相匹配。

要使用不等式来指定范围查询，可以使用$lt和$gt字段（非严格不等式使用$lte和$gte）。例如，要查找时间戳在11月1日午夜之后的所有文档，应该使用下列查询：

```
> db.metrics.find({customer_id:1,ts:{$gte:new Date(2013,11,01)}});
```

要匹配多个选项中的任意一个，可以在查询中使用$in字段和一个数组：

```
> db.metrics.find({customer_id:{$in:[1,2,3,]}});
```

度量集合的查询性能

MongoDB有一个方法explain。可以对查询调用该方法来显示特定查询将要执行的查询计划。这对于在MongoDB环境中调试性能瓶颈特别有用。

例如，下面的查询搜索特定客户度量，它的查询计划说明，在当前索引机制下将会需要一次扫描：

```
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {

  }
}
```

不过，如果使用了时间戳，用于实现生存时间特性的索引会被用来提高性能：

```
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 2,
  "indexBounds" : {
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
  }
}
```

当然，仍然需要对所有客户进行一次扫描。如果客户的数量较小，这不成问题，但我们希望处理的是海量的客户。这时，向客户字段加入索引似乎就是一个好主意了：

```
> db.metrics.ensureIndex({customer_id:1});
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
  }
}
```

可惜的是，MongoDB在一次给定查询中只能使用一个索引，因此加入customer_id索引对这个查询没什么用。要提高性能，必须加入组合索引：

```
> db.metrics.ensureIndex({customer_id:1,ts:1});
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor customer_id_1_ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 0,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 2,
  "indexBounds" : {
    "customer_id" : [
      [
        1,
        1

      ]
    ],
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
  }
}
```

当然，原来的客户查询仍然使用旧有的customer_id索引：

```
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BtreeCursor customer_id_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 3,
  "indexBounds" : {
    "customer_id" : [
      [
        1,
        1
      ]
    ]
  }
}
```

这个查询其实也可以使用组合查询，因此单个customer_id索引现在就是冗余的，应该丢弃：

```
> db.metrics.dropIndex("customer_id_1");
{ "nIndexesWas" : 4, "ok" : 1 }
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BtreeCursor customer_id_1_ts_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
```

```
    "millis" : 0,
    "indexBounds" : {
      "customer_id" : [
        [
          1,
          1
        ]
      ],
      "ts" : [
        [
          {
            "$minElement" : 1
          },
          {
            "$maxElement" : 1
          }
        ]
      ]
    }
  }
```

与SQL数据库类似，MongoDB也提供对查询中的数据进行分组和聚集的功能。刚开始时，聚集功能由group（）或mapReduce（）命令支持，但从2.2版本开始，MongoDB还支持优化的aggregate（）命令。

与SQL不同的是，MongoDB的流水线（pipeline）命令使用流水线方式计算结果，采用大量过滤和分组命令来得到最终结果。用实际运行的例子来解释最易理解，因此我们首先用一些示例数据构建一个集合：

```
> abc = ['A','B','C','D','E','F','G','H','I','J','K','L',
  'M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
> db.createCollection("aggtest");
> for(var i=0;i<1000;i++) {
... db.aggtest.insert({
... first:abc[Math.floor(Math.random()*abc.length)],
... second:abc[Math.floor(Math.random()*abc.length)],
... count:Math.floor(1000*Math.random())
... });
... }
> db.aggtest.find({})
{ "_id" : ObjectId("53213bc8ae5fcad63d0563e9"),
  "first" : "S", "second" : "W", "count" : 762 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ea"),
  "first" : "E", "second" : "V", "count" : 381 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563eb"),
  "first" : "Q", "second" : "O", "count" : 143 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ec"),
  "first" : "C", "second" : "I", "count" : 601 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ed"),
  "first" : "B", "second" : "C", "count" : 413 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ee"),
  "first" : "M", "second" : "D", "count" : 790 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ef"),

  "first" : "S", "second" : "Q", "count" : 699 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563f0"),
  "first" : "A", "second" : "M", "count" : 615 }
... other output omitted
Type "it" for more
```

聚集计算流水线的第一个阶段通常是过滤，其作用类似于SQL语句的WHERE子句。可以根据$match语句来识别过滤阶段，在下面的例子中，选择“first”元素的值为“A”的所有元素：

```
> db.aggtest.aggregate([{$match:{first:"A"}}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563f0"),
      "first" : "A",
      "second" : "M",
      "count" : 615
    },
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563f4"),
      "first" : "A",
      "second" : "F",
      "count" : 806
    },
    {
      "_id" : ObjectId("53213bc8ae5fcad63d056402"),
      "first" : "A",
      "second" : "Q",
      "count" : 377
    },
...more content omitted...
  {
      "_id" : ObjectId("53213bc9ae5fcad63d0567c5"),
      "first" : "A",
      "second" : "G",
      "count" : 769
    }
  ],
  "ok" : 1
}
```

其他过滤选项包括$limit和$skip。$limit多用于测试，作为初始过滤器，它负责限制进入聚集计算的元素数量，如下例所示：

```
> db.aggtest.aggregate([{$limit:1}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563e9"),
      "first" : "S",
      "second" : "W",
      "count" : 762
    }
  ],
  "ok" : 1
}
```

$limit命令更常用于分组和排序操作之后，用来对向用户的输出加以限制。$skip命令与之类似，它会忽略进入过滤器的一定数量的文档。$skip命令与$limit命令经常结合起来在分组之后使用，用来实现分

页：

```
> db.aggtest.aggregate([{$skip:10},{$limit:1}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563f3"),
      "first" : "M",
      "second" : "E",
      "count" : 437
    }
  ],
  "ok" : 1
}
```

在流水线中应用过滤命令之后，就应使用分组管理命令。该类命令最常用的是$group操作符，它指定一个标识符字段和一些累加器。例如，要对各个“first”字段值的“count”字段求和，流水线的声明应如下所示：

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}}]);
{
  "result" : [
    {
      "_id" : "V",
      "total" : 18224
    },
    {
      "_id" : "Y",
      "total" : 15299
    },
    {
      "_id" : "D",
      "total" : 20929
    },
    {
      "_id" : "I",
      "total" : 13257
    },
    {
      "_id" : "N",
      "total" : 16601
    },
...other output omitted...
    {
      "_id" : "E",
      "total" : 21444
    }
  ],
  "ok" : 1
}
```

前面的例子使用$sum累加器，其实还有许多其他的累加器可供使

用。其中包括计算平均值的$avg、计算最小值的$min和计算最大值的$max。_id字段可以包含多个值，因此也支持在多个字段上的分组，但分组的输出通常是无序的。要对输出进行排序，可以对输出应用$sort操作：

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}},{$sort:{total:-1}}]);
{
  "result" : [
    {
      "_id" : "H",
      "total" : 27990
    },
    {
      "_id" : "M",
      "total" : 27188
    },
    {
      "_id" : "L",
      "total" : 25070
    },
    {
      "_id" : "A",
      "total" : 24148
    },
    {
      "_id" : "O",
      "total" : 21865
    },
...other output omitted...
    {
      "_id" : "Q",
      "total" : 12831
    }
  ],
  "ok" : 1
}
```

前面曾提到，这可以通过将$skip和$limit命令结合起来实现。例如，要找到前三个元素：

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}},{$sort:{total:-1}},{$limit:3}]);
{
  "result" : [
    {
      "_id" : "H",
      "total" : 27990
    },
    {
      "_id" : "M",
      "total" : 27188
    },
    {
      "_id" : "L",
      "total" : 25070
    }
  ],
  "ok" : 1
}
```

6.2.2.11　复制

复制是MongoDB的重要主题。在该数据库的早期版本中，都声称单机的持久性是不可信赖的，因此非常强调使用复制来确保数据持久性。虽然2.0及以后的版本所支持的journaling写功能可以提高单机持久性，但是多机复制仍然是MongoDB持久性的优先策略。

为了应对复制问题，MongoDB使用了称为副本集（Replica Set）的系统。与Kafka的同步副本类似，该方法本质上仍然是主-从架构，但主机器是由仲裁的集合成员选出来的，而不是显式指定的。当前的主机器称为Primary，它通过心跳机制跟踪所有其他成员（称为Secondary）。所有的写操作都要在Primary上执行，但读操作可以在副本集的任何服务器中进行。

注意　MongoDB中的复制是异步的。从Primary读总是会返回最新的数据，但从Secondary读则不一定。令情况更为复杂的是，MongoDB的开

发人员还允许显式地设置复制延迟。这样就会创建故意滞后于Primary一段特定时间的副本。在某些类型的灾难恢复场景下，这种做法会派上用场。为了防止客户端不小心从这些滞后副本中读取数据，MongoDB还可以隐藏副本集中的副本，使其无法被客户端获取。

除了维护与Primary之间的心跳连接，Secondary之间也会维持心跳连接。如果与Primary失去联系超过10秒，Secondary会自行选出一个新的Primary，从而支持故障切换，该切换通常是自动进行的。这可能导致发生下面的情况：在Primary与集群的其他机器失去联系前，就将已经写到Primary的数据回滚。

6.2.2.12　分片

除了复制，MongoDB还支持对存储在集合中的数据（除了固定集合）进行自动平衡的分片。其实现非常类似于twemproxy构建Redis集群所用的方法，唯一的不同是，随着时间的推移，数据可能会在分片之间移动。

MongoDB分片的实现致力于从未分片环境平稳过渡到分片环境。如果你开始使用MongoDB，通常情况下我们的建议是，在刚开始时使用常规副本集中的未分片数据集，如果因为数据集增长，工作集合（working set）接近于单台服务器的可用RAM大小，再加入分片。

这种方法之所以奏效，因为MongoDB分片其实就是副本集。为了实现分片，MongoDB引入了名为mongos的辅助服务器。Mongos与一个配置服务器相结合，就可以作为应用和数据库分片之间的中介，而该配置服务器实际上就是另一个MongoDB副本集。mongos负责分发查询和整理返回客

户端的结果。在这个意义上，mongos与用来将Redis和Memcached实例分片的twemproxy服务器非常相似。对分片的MongoDB集群来说，在任意时刻可以有任何数量的mongos服务器运行。常见的策略是，在每个应用服务器上运行一个mongos过程，从而简化应用配置。

MongoDB服务器与twemproxy的不同点在于，它还负责集群的重新平衡。许多其他分片方法依赖预先确定的分片键，MongoDB不同，它试图在集群中自动平衡数据和负载。重新平衡是由balancer进程处理的，该进程由加入集群的某一个mongos服务器初始化。一般来说，该进程是自行启动的。当mongos检测到集群资源中有足够的不平衡性时，该进程才会启动。但是，由于重新平衡集群会加重系统的负荷，因此可以通过配置，使重新平衡进程只在特定的时间窗口内运行，在该窗口中时系统的预期负荷比较低（例如对多数互联网应用来说，凌晨时分就是这样的时间窗口）。

如果打算使用MongoDB的分片功能，首先必须建立若干配置服务器。这些配置服务器的作用与ZooKeeper对Kafka起到的作用一样，也负责提供配置的元数据。最重要的是，它们将包含键在分片中是如何分布的信息。MongoDB不使用像ZooKeeper那样独立的应用，而是使用专门配置的MongoDB数据库，将它作为配置管理器。向mongod的命令行参数中加入--configsvr，就可以激活配置服务器：

```
$ mkdir configdb
$ ./mongod --configsvr --dbpath ./configdb
...
[initandlisten] command local.$cmd command: {
  create: "startup_log",
  size: 10485760,
  capped: true
} ntoreturn:1 keyUpdates:0  reslen:37 168ms
[initandlisten] ******
[initandlisten] creating replication oplog of size: 5MB...
[initandlisten] ******
[websvr] admin web console waiting for connections on port 28019
[initandlisten] waiting for connections on port 27019
```

注意，配置服务器启动的端口与常规的MongoDB服务器端口不同。这样，如果需要，就可以在同一台机器上同时运行常规的分片服务器和配置服务器。这些服务器负载很轻，因此如果让它们与其他服务器共享资源，这通常不会有什么问题。

如果要尝试在开发环境的机器上运行分片服务器，那就要关闭所有正在运行的mongod进程。Mongos服务器与mongod的运行端口相同，并且由于客户端将与mongos连接，故而通常修改mongod端口要容易些。如果要单台机器上测试mongos，应在启动配置服务器之后，在另一个端口启动mongod服务器：

```
$ ./mongod  --dbpath ./db/ --port 27018
 [initandlisten] MongoDB starting : pid=9653 port=27018 dbpath=./db/
   64-bit
 [initandlisten]
 [initandlisten] ** WARNING: soft rlimits too low.
  Number of files is 256, should be at least 1000
 [initandlisten] db version v2.4.8
 [initandlisten] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
 [initandlisten] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64
  BOOST_LIB_VERSION=1_49
 [initandlisten] allocator: system
 [initandlisten] options: { dbpath: "./db/", port: 27018 }
 [initandlisten] journal dir=./db/journal
 [initandlisten] recover : no journal files present, no recovery needed
 [websvr] admin web console waiting for connections on port 28018
 [initandlisten] waiting for connections on port 27018
```

然后就可以在常规的端口上启动mongos服务器，参数指向运行于同一台机器上的配置服务器：

```
$ ./mongos --configdb localhost
 [mongosMain] MongoS version 2.4.8 starting: pid=9657 port=27017 64-bit
  (--help for usage)
 [mongosMain] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
 [mongosMain] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
 [mongosMain] options: { configdb: "localhost" }
 [mongosMain] starting upgrade of config server from v0 to v4
 [mongosMain] starting next upgrade step from v0 to v4
 [mongosMain] writing initial config version at v4
  Mon Dec  2 20:07:18.120
 [mongosMain] upgrade of config server to v4 successful
 [Balancer] about to contact config servers and shards
 [websvr] admin web console waiting for connections on port 28017
 [Balancer] config servers and shards contacted successfully
 [mongosMain] waiting for connections on port 27017
```

在生产环境的系统中，应该至少有三个配置服务器。应该向mongos提供一个服务器列表（列表中配置服务器数量要大于1，以支持故障切换），列表以逗号间隔，并且至少应包含一部分配置服务器。单台配置服务器只能用于测试目的。

一旦mongos服务器运行起来，就可以用mongo客户端来配置集群。Sh.addShard函数用来向集群加入分片（该命令只有当连上mongos服务器时才可以使用）。如果该服务器以单机运行，那就只需要主机名：

```
mongos> sh.addShard("localhost:27018")
{ "shardAdded" : "shard0000", "ok" : 1 }
```

要加入一个副本集，只需要加入属于该副本集的一个服务器（旧版本的MongoDB需要以逗号间隔列表的形式，将所有服务器都加入进去）。

```
mongos> sh.addShard("replset1/localhost:27018")
{ "shardAdded" : "shard0000", "ok" : 1 }
```

最后，对数据库激活分片：

```
mongos> sh.enableSharding("mine");
{ "ok" : 1 }
```

对数据库中特定集合来说，分片的控制有更细粒度的选择，但上面所讲的是最基本的使用案例。有了分片，MongoDB就可以支持工作集合远超单台机器所能承受的数据库。

6.2.3 Cassandra

Cassandra数据库系统实现的特性与Amazon的DynamoDB（Cassandra最初的作者之一曾参与DynamoDB项目）及Google的BigTable很类似。

可惜的是，除了实现方面类似于BigTable，Cassandra的“查询语言”也与BigTable很像。它的查询语言基于内部的Thrift数据结构，使用起来很

烦琐。多数开发人员不熟悉这种语言。这让Cassandra得到了难使用和难维护的坏名声，一度走进“死胡同”。

Cassandra最初的开发者是Facebook，它在实现Facebook的即时通讯特性时“抛弃”了Cassandra，转而使用HBase，这导致业界对Cassandra更是大加诟病。当然，与这件事关系最大的是一致性模型问题，即时通讯应用需要是强一致性的。如果在用户的应用所使用的集群中，服务器的状态恰巧与上一次调用的最后的服务器状态不同，用户绝不能“丢失”消息。Cassandra的数据模型是最终一致性的，因此对该应用来说，它确实不是好的选择。HBase是强一致性的，尽管自身存在一些问题，但它的强一致性使其更适合这类应用。而对于多数实时分析应用来说，最终一致性是可以容忍的，因为数据是流动的，一致性服务器的不断变化的集合更像是应用的延迟。

Cassandra查询语言（Cassandra Query Language，CQL）的引入对Cassandra的缺陷做出了很大弥补。CQL 3.0基本上将所有最初的类BigTable结构抽象为多数开发人员熟悉的类SQL语言。CQL语言有一些限制，比如聚集函数上的限制，但Cassandra的设计支持对表进行广泛的非规范化（denormalization）。这增加了存储开销，引入了写时的聚集计算代价，但在Cassandra这样的横向扩展（scale-out）架构中，这不是什么大问题。

本节介绍Cassandra的架构和数据模型。Cassandra与受DynamoDB和BigTable启发的其他数据库（例如HBase和Voldemort）类似，本节介绍的许多概念也适用于这些数据库。本节还涵盖Cassandra所使用的内部数据

模型，这主要是历史原因，因为CQL将许多这些内部数据模型抽象成自己的数据模型，而两者有时并不完全兼容。

6.2.3.1 服务器架构

Cassandra服务器架构是“无主机器”节点集群，其目标是消除单点故障，以及支持线性扩展。为此，Cassandra的数据管理使用分布式哈希表（Distributed Hashing Table，DHT）模型。在该模型中，为每行数据指定一个分区键。这个键定义由哪个服务器存储特定的一份信息。为了提高数据持久性，Cassandra还可以通过一致性哈希将数据复制到其他节点。

Cassandra的早期版本（1.2之前的版本）使用基于服务器的一致性哈希技术，该技术需要计算token并指定给每个节点，从而告诉节点它要存储的哈希范围，这都颇要一番维护工作。较新版本的Cassandra引入了虚节点的概念，称为“vnode”。虚节点将哈希范围分为小段，这些小段会被随机分布到节点中。

每个虚节点都对包含在分区中的数据维护自己的索引，以及一个称为Bloom Filter（第10章会详细讨论）的专门数据结构，该数据结构用来快速确定查询是否需要特定虚节点的数据。

集群中的节点通过一种名为“gossip”的点对点通信协议来维护其他服务器的地图。通过调用该协议，每个服务器与集群中最多三个其他服务器交换状态信息，基本上每秒钟交换一次。该信息中还包含其他服务器的状态信息，因此通过与对应的节点交换状态，给定的节点很快就会学习到整个集群的拓扑。

除了gossip协议，还会用到“snitch”协议来确定本地网络拓扑。该协议帮助Cassandra优化它的请求路由，并发现处于活跃状态但又是降级（degraded）状态的节点。在实际应用中，会由于各种原因出现“sick”节点，这种情况很常见。针对各种部署有各种不同类型的优化snitch。例如，在Amazon的EC2上运行的集群有两种snitch：EC2Snitch和EC2MultiRegionSnitch。前者的设计目标是，当集群建立于单个地域（region）的多个可用的区域（zone）上（这种情况最常见）时，能够良好运行。后者的设计目标是优化跨地域的集群。

由于每个节点都会通过gossip和snitch获得对集群拓扑的完整理解，因此每个节点都可以作为查询服务器。当客户端连接集群时，可以选择自己钟意的任何节点，然后选中的节点就成为该客户端的协调节点，一直到连接关闭。这基本上相当于，所有的MongoDB服务器都驻留有mogod和mongos。

6.2.3.2　建立集群

从软件角度来看，建立Cassandra集群比较简单，因为各服务器基本上是相同的。为了获得较好的性能，建议你使用RAM为8GB到16GB的多核机器。如果Cassandra中的堆（heap）很大，那么实际上性能会因此下降，因为需要进行垃圾回收。通常推荐的堆大小是8GB左右（取决于可用的RAM）。在Amazon EC2中，这对应大实例和超大实例。

Cassandra服务器应该有尽可能快的硬盘。如果可能的话，最好使用固态硬盘（SSD），它能很好地与Cassandra的访问模式相匹配。如果没有SSD，通过RAID0合并的硬盘组也是不错的选择。Cassandra与Kafka和

HDFS类似，它具备使用JBOD（Just a Bunch of Disks）配置下的多个硬盘的能力，但使用RAID0能达到更高的性能。使用RAID0配置还能消除磁盘间数据分布的不平衡。

不建议对Cassandra使用网络附加存储（Network Attached Storage，NAS），除非作为备份介质。NFS或弹性块存储（Elastic Block Storage，EBS）等网络存储系统会竞争网络I/O资源，导致性能下降。

建立服务器本身是很简单的。可以在Apache Cassandra网站（http://cassandra.apache.org）上找到对Debian系统的安装说明。Datastax公司是Cassandra的商业提供商和活跃的代码贡献者，他们也提供了针对Windows和OS X等一系列平台的Cassandra发布包。如果是要建立自己的发布包的用户，也可以从Apache网站下载二进制tarball。在正确安装Java的前提下，多数现代类Unix操作系统都应该能直接使用该tarball。

对于Cassandra来说，Java 7或更高版本是合适的。用Java 6旧版本启动Cassandra这种情况仍然很常见，但会导致如下错误：

```
$ java -version
java version "1.6.0_65"
Java(TM) SE Runtime Environment (build 1.6.0_65-b14-462-11M4609)

$ ./bin/cassandra -f
xss =  -ea -javaagent:./bin/../lib/jamm-0.2.5.jar
  -XX:+UseThreadPriorities
  -XX:ThreadPriorityPolicy=42
  -Xms1024M -Xmx1024M
  -Xmn256M -XX:+HeapDumpOnOutOfMemoryError -Xss256k
Exception in thread "main"
java.lang.UnsupportedClassVersionError
org/apache/cassandra/service/CassandraDaemon :
Unsupported major.minor version 51.0
```

主要版本51指的是Java 7（Java 6是50，Java 8是52）。如果安装了Java 7或更高版本，就可以用cassandra或cassandra–f命令来启动Cassandra，从而在前台运行服务器。

6.2.3.3 配置选项

Cassandra 2.0的多数配置选项是可以“开箱即用”的，但也有一些配置选项要经常修改来适应特定的Cassandra安装。你可以在conf/目录找到这些选项。Cassandra.yaml指定了集群名称，该名称默认为“Test Cluster”：

```
# The name of the cluster. This is mainly used to prevent machines in
# one logical cluster from joining another.
cluster_name: 'Test Cluster'
```

默认情况下，Cassandra会将数据写入/var/lib/cassandra，因此必须以适当的权限创建该目录。要改变这个选项，需要修改cassandra.yaml文件中三个部分，它们指定了数据输出的目录：

```
# Directories where Cassandra should store data on disk.  Cassandra
# will spread data evenly across them, subject to the granularity of
# the configured compaction strategy.
data_file_directories:
    - /var/lib/cassandra/data

# commit log
commitlog_directory: /var/lib/cassandra/commitlog

# saved caches
saved_caches_directory: /var/lib/cassandra/saved_caches
```

对于多核集群来说，需要设置listen_address和rpc_address。第一个参数listen_address是节点用来进行gossip协议通信的网络接口，第二个参数rpc_address用于客户端连接。将这两者都置为空也没问题，但有时有必要明确指定要使用的网络接口地址。默认情况下listen_address设置

为“localhost”，这个值通常不管用：

```
# Address to bind to and tell other Cassandra nodes to connect to. You
# _must_ change this if you want multiple nodes to be able to
# communicate!
#
# Leaving it blank leaves it up to InetAddress.getLocalHost(). This
# will always do the Right Thing _if_ the node is properly configured
# (hostname, name resolution, etc), and the Right Thing is to use the
# address associated with the hostname (it might not be).
#
# Setting this to 0.0.0.0 is always wrong.
listen_address:
# The address to bind the Thrift RPC service and native transport
# server -- clients connect here.
#
# Leaving this blank has the same effect it does for ListenAddress,
# (i.e. it will be based on the configured hostname of the node).
#
# Note that unlike ListenAddress above, it is allowed to specify 0.0.0.0
# here if you want to listen on all interfaces, but that will break
# clients that rely on node auto-discovery.
rpc_address:
```

最后，多节点集群服务器需要一系列种子服务器，用来将它自己引导进集群。这个服务器列表并不需要包括集群中的每个服务器，但要有足够的数量，以保证当该服务器启动时，列表中至少有一个服务器是可用的。只需要将seeds修改为以逗号间隔的服务器列表：

```
# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and
    # learn the topology of the ring.  You must change this if you are
    # running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "127.0.0.1"
```

6.2.3.4 CQL：Cassandra的查询语言

CQL是开发所有新Cassandra应用的推荐机制。它实现了结构化查询语言（Structured Query Language，SQL）的一个子集，因此许多开发人员会对它比较熟悉。但是，许多SQL特性它都没有实现，并且Cassandra表的结构会让许多开发人员感到很突兀。

可以通过三种不同的机制来运行CQL命令。第一种机制是名为cqlsh的CQL shell，它是Cassandra自带的Python应用。它与cassandra-cli不同，后者使用Thrift API。你可以在Cassandra服务器二进制文件的目录中找到它。第二种机制是通过“本地”CQL驱动。新版本的Cassandra实现了只支持CQL语句的协议，该协议与关系数据库使用的协议类似。最后，通过使用旧客户端，可以利用Cassandra的Thrift协议用execute_cql3_query远程过程调用（RPC）命令来执行CQL命令。

6.2.3.5 键空间和列族

所有的Cassandra数据都存放在键空间中。键空间是一种名字空间，基本上等同于MongoDB中的数据库或关系数据库中的模式。创建键空间时会定义复制策略和复制因子。复制因子就是存放在集群中的任何一份给定数据的副本数量。复制策略定义了这些副本的分布方式。开始时，需要用cqlsh创建metrics键空间，并将复制因子置为1：

```
cqlsh> CREATE KEYSPACE metrics WITH REPLICATION =
   {'class':'SimpleStrategy','replication_factor':1};
```

以后可以使用ALTER KEYSPACE命令来修改这些设置。特别如果在有多个节点的生产环境中使用集群时，“class”定义的复制策略就应该是

NetworkTopologyStrategy。该策略支持多数据中心，而例子中使用的SimpleStrategy仅支持单数据中心。即使只使用了一个物理的数据中心，常见的做法还是定义集群的多个“数据中心”，各数据中心维护不同的负载。例如，一个“数据中心”可以专门用来接收实时读写的随机访问，而另外一个“数据中心”专门用于扫描频繁的Map-Reduce式处理。这往往是用Hadoop来完成的，因为Cassandra可以替代Hadoop的本地HDFS存储层。

如果要创建表，通过USE命令切换到metrics键空间，然后执行CREATE TABLE来存储一些时间序列度量：

```
cqlsh> USE metrics;
cqlsh:metrics> CREATE TABLE counts (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  PRIMARY KEY (customer_id,metric,ts)
);
```

没有必要选择键空间，可以使用点表示法来访问表目录。用metrics.counts就可以访问上面例子中的表。

这个表包含客户标识符、度量名称和时间戳。所有这三者构成了主键，因为客户、度量和时间三者联合起来唯一地定义了一条数据。在Cassandra中，列序列化的结果是对应其类型的字节数组，相反的过程就是反序列化。这些类型与关系数据库中的类型类似。表6-2总结了这些类型。

表6-2　CQL数据类型

名　称	描　述
ASCII	编码为 US-ASCII（0 ～ 127 的字符）的字符串
BIGINT	64 位有符号长整数（等价于 Java long）
BLOB	任意字节数组
BOOLEAN	True 或 false 布尔值
COUNTER	64 位分布式计数器值。在 Cassandra 有特殊处理
DECIMAL	可变精度小数
DOUBLE	IEEE-754 兼容的 64 位浮点数
FLOAT	IEEE-754 兼容的 32 位浮点数
INET	表示 IPV4 地址或 IPV6 地址的结构
INT	32 位有符号整数（等价于 Java int）
LIST<?>	一个值列表，其中“?”是合法的 CQL 类型。例如，LIST<TEXT> 是 UTF-8 字符串的列表。集合中不应该包含其他集合，因此 LIST<LIST<TEXT>> 是错误的
MAP<?,?>	CQL 类型的关联数组。与 LIST 类型类似，MAP 有两个合法的 CQL 标量类型
SET<?>	SET 集合类型与 LIST 类型类似，但它只存储唯一值
TEXT	UTF-8 编码的字符串
TIMESTAMP	64 位整数（BIGINT）编码的时间戳，为 UNIX 新纪元以来的毫秒值
UUID	全局唯一标识符。对于 Windows 用户来说，UUID 与 GUID 是相同的
TIMEUUID	带有时间戳分量的类型 1 UUID。该类型允许多个客户端插入时间戳而不会冲突。可以在这种类型的列上进行时间范围查询。当进行插入时，NOW( ) 函数被用来生成 TIMEUUID
VARCHAR	UTF-8 编码的字符串
VARINT	任意精度的整数值

来源：http://www.datastax.com/documentation/cql/3.1/webhelp/index.html#cql/cql_reference/cql_data_types_c.html#concept_ds_wbk_zdt_xj

计数器列

在Cassandra中，原子计数器列不能与列族中的其他非键列组合。计数器列可以与其他计数器列组合。如果定义前面的表时去掉了主键中的度量，会导致错误：

```
cqlsh:metrics> CREATE TABLE counts (
  customer_id INT PRIMARY KEY,
  metric TEXT,ts TIMESTAMP,
  value COUNTER
);
Bad Request: Cannot add a counter column (value) in
  a non counter column family
```

可以在前面的表中定义第二个计数器：

```
cqlsh:metrics> CREATE TABLE counts_2 (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  value_2 COUNTER,
  PRIMARY KEY (customer_id,metric,ts));
```

如果由于某种原因需要计数器列和非计数器列，最简单的方法就是定义两个表，并使用批量更新语句同时更新它们。

在Cassandra内部，用列族（column family）数据结构来表示该数据表。尽管看起来似乎包含了大量行，但实际上列族对每个customer_id只包含一行，对每个metric和ts的组合则包含了大量列。这是唯一需要注意的，因为给定一行，对应的列的数量是有限的：20亿列或2GB的存储。

在某些时间序列实现中，列的数量很容易达到这个规模。为了解决这个问题，Cassandra允许将多个键用作行的标识符。这样做的缺点是行的键（row key）也被用来在Cassandra集群中对数据进行分区。这意味着所有的查询，包括插入或更新，都必须包含行键的所有元素。

如果查询始终包含customer_id和metric，将customer_id和metric字段合并会创建由customer_id：metric的组合所标识的行，并有一个时间戳列：

```
cqlsh:metrics> CREATE TABLE counts_composite (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  value_2 COUNTER,
  PRIMARY KEY ( (customer_id,metric) ,ts)
) WITH CLUSTERING ORDER BY (ts DESC);
```

这里加入了CLUSTERING ORDER命令，这是为了告诉Cassandra对每列按降序排序，而不是按时间戳列的自然序（即升序）排序。

与多数关系数据库类似，表创建完成之后，你可以用ALTER TABLE命令进行修改。最常见的情况是向已有表中加上一列，或者删除已有的列。如果是加入新列，那就不需要对已有的行进行校验。

如果删除已有列，与该列相关的数据最终也会被删除，但只有当发生完整文件合并（major compaction）时才会执行删除。

某些情况下，可以修改列类型，但这不会改变存储已有列数据的字节。如果无法通过新的列类型将数据反序列化，那么查询数据时就会发生错误。例如，如果将TEXT类型的列转换为INT类型的列，那可能就不会得到预期效果。

6.2.3.6　插入和更新数据

要向Cassandra中插入和更新数据，应通过我们熟悉的INSERT和UPDATE CQL语句来完成。这两个语句基本上类似于对应的SQL语句，但也有一些不同之处。

INSERT语句的构成与SQL相同，要注意的是，要更新的所有列都必须

包含在语句中（在任何情况下这都是一个好习惯）。例如，向时间序列表中插入标准规格样式（gauge-style）的值：

```
cqlsh:metrics> CREATE TABLE stats(
  customer_id INT,
  metric TEXT,
  ts TIMEUUID,
  value INT,
  PRIMARY KEY( (customer_id,metric), ts)
) WITH CLUSTERING ORDER BY (ts DESC);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES
  (1,'hits',NOW(),10);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES (
  1,'hits',NOW(),15);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES
  (1,'hits',NOW(),30);
cqlsh:metrics> SELECT * FROM stats;

customer_id | metric | ts                                   | value
------------+--------+--------------------------------------+------
          1 |   hits | 1ae93880-5fb8-11e3-9b3a-a1e3c690259e |    30
          1 |   hits | 19925b10-5fb8-11e3-9b3a-a1e3c690259e |    15
          1 |   hits | 175897b0-5fb8-11e3-9b3a-a1e3c690259e |    10
```

默认情况下，INSERT语句会覆盖具有相同主键的数据。这并不是SQL数据库的常见做法，因此有的人可能感到意外。有的时候这可能会引发问题，因此，从CQL3开始，Cassandra支持称为轻量级事务（Lightweight Transaction）的特性，通过在INSERT语句末尾加入IF NOT EXISTS语句恢复了类似SQL的行为。例如，使用了IF NOT EXISTS语句，created_on字段置为NOW（）的用户表就不会在每次插入后发生改变：

```
cqlsh:metrics> CREATE TABLE users(email TEXT PRIMARY KEY,created_on
  TIMEUUID);
cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW());
cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;
 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:40:24-0800

(1 rows)

cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW());
cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;
 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:41:09-0800

(1 rows)

cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW())
           ... IF NOT EXISTS;
 [applied] | email        | created_on
-----------+--------------+--------------------------------------
     False | byron@domain | 9351e360-5fba-11e3-9b3a-a1e3c690259e

cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;

 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:41:09-0800

(1 rows)
```

Update语句也类似于对应的SQL更新语句，只有一点除外，那就是它们默认会执行一次upsert。正如INSERT会覆盖数据而不是报告失败一样，如果数据集中不存在主键的话，插入操作的执行是隐含在UPDATE语句中的。这对于COUNTER列实际上是非常有用的，因为不能用INSERT语句来创建该列：

```
cqlsh:metrics> DROP TABLE counts;
cqlsh:metrics> CREATE TABLE counts(
  customer_id INT,
  metric TEXT,
  value COUNTER,
  PRIMARY KEY (customer_id, metric)
);
cqlsh:metrics> UPDATE counts SET
  value = value + 1
WHERE customer_id = 1 AND metric = 'test';
cqlsh:metrics> SELECT * FROM counts;

 customer_id | metric | value
-------------+--------+-------
           1 |   test |     1

(1 rows)
```

通过像WHERE语句那样在语句末尾加上一个IF语句，Update语句还支持检查和设置（check-and-set）操作。这样，只有当IF语句的值为真时，更新操作才会成功执行。

INSERT和UPDATE语句还都支持用生存时间（time-to-live）语句设置数据的失效期。只需要在INSERT语句末尾加上USING TTL＜second＞即可。对于UPDATE语句，需要将其放在表名的后面：UPDATE＜table＞USING TTL＜seconds＞。

为了提高性能，可以将insert和update语句打包到BATCH语句。当向许多不同的表写数据，或者处理大量请求时，这样做会极大地提高性能。要使用BATCH语句，只需要以BEGIN BATCH来启动命令，然后像往常一样写入语句，当你准备好提交时，就向语句末尾加入APPLY BATCH。

Cassandra的度量

我们曾在本节的例子中展示过，Cassandra完全可以用原子计数器来

获取度量数据。它缺少的是在服务端进行任意形式的聚集计算的能力。在这里，我们推荐的策略是，使用批量更新，从而在写数据的同时计算出全部聚集。

为了在客户级和系统级都进行聚集计算，可以使用两个表来计算聚集：

```
cqlsh:metrics> CREATE TABLE customer_counts (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  PRIMARY KEY ( (customer_id,metric) , ts) )
WITH CLUSTERING ORDER BY (ts DESC);
cqlsh:metrics> CREATE TABLE system_counts (
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER, PRIMARY KEY ( metric , ts) )
WITH CLUSTERING ORDER BY (ts DESC);
```

为了对所有表进行更新，我们使用了BATCH命令。这并不是事务，但它确实允许Cassandra执行高效的更新：

```
cqlsh:metrics> BEGIN COUNTER BATCH
           ... UPDATE customer_counts
               SET value = value + 1
               WHERE customer_id = 1
               AND metric = 'impressions'
               AND ts = '2013-12-01T00:00:00';
           ... UPDATE system_counts
               SET value = value + 1
               WHERE metric = 'impressions'
               AND ts = '2013-12-01T00:00:00';
           ... APPLY BATCH;
```

6.2.3.7 从Cassandra读取数据

与插入数据类似，如果要从Cassandra读数据，那就使用类似SQL的

SELECT语句。Cassandra没有多少查询选项，因此SELECT语句是相当简单的。

6.3　其他存储技术

本章聚焦于在实时应用中广泛使用的几种不同的持久存储系统。之所以选择这些存储系统，是为了向你展示除了关系数据库之外的实时应用存储系统的选择范围。本节只是简要提及可以使用的其他技术，而不会作具体讨论。

6.3.1　关系数据库

许多关系数据库实现要依靠高性能的数据存储。由事务处理造成的性能损失是在应用层维护的，而不是在数据库本身固有的。某些关系数据库甚至允许访问这些高性能存储。例如，PostgreSQL通过它的hstore模块向外部提供键-值存储，用户可以通过SQL接口对其进行访问。它还支持对键-值存储进行索引。

其他数据库，特别是商用的列存储的设计目标是摄取大量原始数据，并支持对这些数据的高效查询。在没有良好定义的访问模式的情况下，将流式数据放入列存储来访问可能是最佳方法。如果要达到可以接受的数据摄取性能，列存储需要付出一些代价，这也成为列存储的缺点。不过，这往往是一个在查询灵活性和性能之间如何进行权衡的问题。

6.3.2　分布式内存数据网格

各种形式的数据网格已经使用了许多年。数据网格通常是由VMware

和Oracle等著名厂商作为商业产品销售的。为了推广产品，许多数据网格现在有社区版本。只要保证用于测评，就可以免费使用社区版本。Hazelcast就属于这类产品。

数据网格的主要设计目标是提供数据的分布式视图，从该视图看，数据就像是存储在应用的主存储器中一样。数据网格往往使用特定的编程语言，其中Java最流行。还会实现原始集合（collection）的分布式版本，如List和Map的Java接口。

6.4 存储技术的选择

讨论持久化技术时，难免会对哪个技术最好产生争论。类似于哪个文本编辑器或集成开发环境“最好”，这些争论饶有趣味，但归根结底没多大意义。实际上，只存在给定环境下对给定应用效果良好的技术，而不存在绝对的优胜者。

开发人员在特定环境工作之后，可能会成功地将经验推广到其他应用。而只因为它投放市场的时间最短，就将该技术作为最佳选择，尽管客观来说其他技术的特性与应用可能更匹配。为了避免这种偏见，我们既总结经验又吸取教训，给出了在考虑使用特定的存储技术时的一些经验法则。

6.4.1 键-值存储

如果事先知道聚集结构，Redis等键-值持久存储的效果最好。通常来说，这些存储系统也青睐简单查询，因为查询完全是由客户端来管理的。这使得它们最适合交互有限的应用，例如仪表板应用。第7章讨论用Redis构建简单的仪表板应用。

6.4.2 文档存储

根据文档存储的设计特点，当有大量度量值要以自然分组的形式存储时，文档存储是最佳选择。例如，客户可能有针对其业务需要的大量可定

制的度量要存储。

如果查询模式是将文档提取到工作集合，而工作集合本质上是将文档“预热”，从而为后续查询提供更快的访问速度，那么使用文档存储的效果也会很好。

6.4.3　分布式哈希表存储

Cassandra、Voldemort、BigTable和DynamoDB是关系数据库与键-值存储的“混血儿”。类似于键-值存储，当聚集模式已知，从而可以由流处理基础架构进行非规范化处理时，这类存储的性能最好。Cassandra的批量更新特性本质上就是为了支持这种多表更新。

与键-值存储不同的是，它们通常提供高效的范围查询或集合查询，通常还支持二级索引以提高查询性能。在Redis这样的键-值存储中，可以使用有序集合这样的特性来达到同样的目标，但如果查询有些复杂，这样做就很不方便。

与关系数据库不同的是，这类存储对ad hoc聚集计算的支持通常不是很好。它们通常不支持GROUP BY和SUM操作，这都需要客户端进行聚集计算。这使得它们不适合更具探索性的数据环境，除非将其与Map-Reduce环境集成来进行聚集计算。

6.4.4　内存网格

在适合分布式哈希表或键-值存储的场景中，内存网格往往也有不错的

表现。由于所有数据都存放在内存中，内存网格的数据访问和更新通常都很快。但是，这也使得它们比基于磁盘的分布式哈希表（DHT）更昂贵。内存网格多数提供某种查询和索引功能，这一点与DHT方法类似。

当存储与应用紧耦合时，内存网格的表现也不错。例如，它可以用于使纵向扩展架构的遗留应用适应可以从实时流系统获取数据的横向扩展架构。

6.4.5　关系数据库

随着NoSQL存储大行其道，有人声称关系数据库的地位有所动摇。这显然言过其实。关系数据库仍然有用武之地，而且在其擅长的领域仍然具有很好的表现。ad hoc聚集计算和查询就特别适合使用关系数据库。对于列数据库则更是如此，因为列数据库有专门的索引和存储机制来优化许多特有的聚集查询。

在一种情况下，关系数据库表现不佳，那就是数据摄取速度非常快时。关系数据库中的“实时”通常指的是，处理数据流时有几分钟或几小时的延迟，而不是几毫秒。（有一些例外。有几个公司研发了专门用于高性能数据摄取的数据库。）通过较慢的ETL过程（马上就会讨论）来管理时，关系数据库通常效果最好。

6.5 数据仓库

现在，许多业务都有了业务智能基础架构。这种基础架构通常由关系数据库环境组成，用某种抽取-转换-加载（Extract-Transform-Load，ETL）工具来加载。

在大型企业中，通常是由数据库团队用一系列正规处理工具来管理。在小型企业中可能往往使用ad hoc机制，常使用小型MySQL数据库，由临时编写的脚本来加载。在这两种情况下，都存在一些小问题。流数据的速度远超ETL基础架构或数据库的处理速度，因为实时处理系统始终在更新数据。进一步说，当（不是如果）引入了错误时，如果具备从错误中恢复并重新处理数据的能力，那就有助于改善运营效率和正常运行时间。随着针对这种处理任务的现代工具的引入，数据仓库过程的实现会更趋一致，也更容易扩展，即使对无力投资复杂ETL基础架构的较小企业也会从中受益。

6.5.1 将Hadoop作为ETL和数据仓库

自从2007年公开发布以来，Hadoop已经几乎成为开发大规模处理和ETL任务的事实上的标准。尽管这很了不起，但Hadoop的处理模型有很多限制，这是因为它本质上解决的是大规模批处理环境的基本管理问题，即将计算与数据绑定。

在大规模ETL流水线中，对数据所执行的各个操作通常都是非常琐细

的。用数据库术语来说，这些操作通常可以归结为语句序列，包括WHERE子句中的GROUP BY语句，以及SELECT子句中的SUM语句，可能还包括DISTINCT语句。更大的问题是管理机器集群中的数据流程，以支持对数据流的高效处理。

6.5.1.1 从Kafka中摄取数据

Kafka为Hadoop提供了非常简单的数据摄取机制。可惜的是，这个机制有点过于简单，因此很难让人有信心在生产环境中使用。更好的选择是Camus工具，LinkedIn就是使用这个工具执行从Kafka到Hadoop的数据摄取。尽管Camus目前已经被投入到了生产环境，但它仍然处于开源生命周期的早期阶段，因此在撰写本书时，它还没有预包装的库。

默认情况下，Camus假定要处理的数据大体遵从LinkedIn自己的内部数据格式，该格式是基于Avro的。多数人并不在LinkedIn工作，因此通常需要建立一个自定义的导入器来定义怎样处理数据。

开始建立自定义导入器时，首先要从Github检出Campus资源库，这样才能将需要的依赖安装到本地Maven资源库中。本书使用Kafka 0.8，该版本还没有合并到主分支中：

```
$ git clone https://github.com/linkedin/camus
Cloning into 'camus'...
remote: Counting objects: 2539, done.
remote: Compressing objects: 100% (991/991), done.
remote: Total 2539 (delta 774), reused 2393 (delta 661)
Receiving objects: 100% (2539/2539), 37.99 MiB | 950.00 KiB/s, done.
Resolving deltas: 100% (774/774), done.
Checking connectivity... done
$ cd camus/
$ git checkout camus-kafka-0.8
Branch camus-kafka-0.8 set up to track remote branch camus-kafka-0.8
from origin.
Switched to a new branch 'camus-kafka-0.8'
$ mvn install
```

在本地安装了Maven包之后，就可以在自定义的loader项目的Maven文件中使用该包了：

```
<dependencies>
    <dependency>
      <groupId>com.linkedin.camus</groupId>
      <artifactId>camus-api</artifactId>
      <version>0.1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.linkedin.camus</groupId>
      <artifactId>camus-etl-kafka</artifactId>
      <version>0.1.0-SNAPSHOT</version>
    </dependency>
    <dependency>

     <groupId>org.apache.hadoop</groupId>
     <artifactId>hadoop-core</artifactId>
     <version>1.0.3</version>
   </dependency>
</dependencies>
```

Camus.properties文件负责控制Camus的加载过程，这个文件位于自定义loader的jar文件中。你可以从camus-example项目中复制该文件，或者使用本书的项目源代码中的版本。如果使用的是来自camus-example的文件，那就需要修改一些条目，这样才能使用自定义loader。

这里假定数据是JSON格式。假定事件的时间戳位于ts字段中，是自Unix新纪元开始的毫秒数。在CamusWrapper这个类中，会提取该时间戳，返回解析后的JSON格式数据以及该时间戳。Camus会使用这个类来移动数据：

```
public class JSONMessageDecoder
  extends MessageDecoder<byte[], ObjectNode> {

  private final ObjectMapper mapper = new ObjectMapper();

  @Override
  public CamusWrapper<ObjectNode> decode(byte[] arg0) {
    ObjectNode json;
    try {
      json = mapper.readValue(new String(arg0), ObjectNode.class);
    } catch (IOException e) {
      throw new MessageDecoderException("Unable to parse event");
    }
    return new CamusWrapper<ObjectNode>(json,
     json.get("ts").getValueAsLong());
  }
}
```

要使用这个类，那就要确保正确设置了camus.message.decoder.class属性：

```
# Concrete implementation of the Decoder class to use
camus.message.decoder.class=wiley.streaming.camus.JSONMessageDecoder
```

一旦数据被安全解码，就必须将其写到文件中。这里会用到SequenceFile。这些文件会存储键和值，因此键将包含事件在Kafka中的位置信息：主题、分区和偏移量，而值则是再编码为文本的JSON格式数据，再编码的目的是为进一步处理提供便利：

```
public class JSONRecordWriter implements RecordWriterProvider {

  @Override
  public RecordWriter<IEtlKey, CamusWrapper> getDataRecordWriter(
      TaskAttemptContext context, String filename, CamusWrapper arg2,
      FileOutputCommitter comitter) throws IOException,
        InterruptedException {
    Configuration conf = context.getConfiguration();
    Path file = new Path(comitter.getWorkPath(),
      EtlMultiOutputFormat.getUniqueFile(context, filename,
       getFilenameExtension()));
    CompressionCodec codec = null;
    SequenceFile.CompressionType type =
      SequenceFile.CompressionType.NONE;


  if(FileOutputFormat.getCompressOutput(context)) {
    type = SequenceFileOutputFormat.getOutputCompressionType(
      context
    );
    Class<?> klass =
SequenceFileOutputFormat.getOutputCompressorClass(context,
 DefaultCodec.class);
    codec = (CompressionCodec) ReflectionUtils.newInstance(klass,
      conf);
  }
  final SequenceFile.Writer writer = SequenceFile.createWriter(
      file.getFileSystem(conf),
      conf,
      file,
      Text.class, Text.class,
      type,
      codec);
  return new RecordWriter<IEtlKey,CamusWrapper>() {
    Text key   = new Text();
    Text value = new Text();


    @Override
    public void close(TaskAttemptContext arg0) throws IOException,
        InterruptedException {
```

```
          writer.close();
        }

        @Override
        public void write(IEtlKey arg0, CamusWrapper arg1)
            throws IOException, InterruptedException {
          key.set(arg0.getTopic()+"\t"+arg0.getPartition()+
            "\t"+arg0.getOffset());
          if(arg1.getRecord() instanceof ObjectNode) {
            ObjectNode node = (ObjectNode)arg1.getRecord();
            value.set(node.toString());
          }
        }

      };
    }

    @Override
    public String getFilenameExtension() {
      return ".json";
    }

}
```

通过修改camus.properties，使用相应的类，就可以激活Camus：

```
etl.record.writer.provider.class=wiley.streaming.camus.
  JSONRecordWriterProvider
```

6.5.1.2 从Flume摄取数据

Flume支持将Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）的数据摄取作为它的一个sink。例如，下列代码使用名为agent99的agent将数据写入与早前的Camus例子相同的目录：

```
agent99.channels = channel1
agent99.sinks = kitchen
agent99.sinks.kitchen.type = hdfs
agent99.sinks.channel = channel1
agent99.kitchen.hdfs.path = /events/%y%m%d%H%M
agent99.kitchen.hdfs.round = true
agent99.kitchen.hdfs.roundValue = 5
agent99.kitchen.hdfs.roundUnit = minute
agent99.kitchen.hdfs.useLocalTimeStamp = false
```

在这个例子中，要求Hadoop分布式文件系统HDFS sink创建路径时使用的本地时间戳向下取整为5分钟的时间间隔。要得到该时间戳，就要查找事件本身的时间戳头部。

6.5.1.3　事件与处理时间

在这类摄取过程中使用事件的时间戳是诱人的，它可以将事件干净利落地放入目录，从而使后续分析更加容易。

可惜的是，这样做会使实现连续的处理流水线变得复杂得多。在流数据处理环境中，数据可能是乱序到达的。这意味着我们必须对摄取序列之后所有数据结束于哪里进行记录。通过导入时间来保持数据顺序，这样当数据导入时就不容易出现混乱。

记录导入时间之所以重要，原因在于当发生错误时，导入时间就是问题的关键。如果处理流水线中断，导入时间（而不是数据的生成时间）就确定了应该处理哪些数据。而对于多数恢复和维护场景来说，事件的时间都无关紧要，只是对于要查看历史数据的用户来说，可以让查看过程更容易而已。

要在Flume中使用本地时间戳，只需要将配置中的useLocalTimeStamp参数改为true。

如果要在Camus中使用本地时间戳，那就要复杂一些。最简单的方式是在作业之前基于当前时间选择一个输出目录。可以传入一个配置参数，然后由自定义的Partitioner类来读取：

```java
public class BatchPartitioner implements Partitioner {

  String batch = null;

  @Override
  public String encodePartition(JobContext context,
    IEtlKey etlKey) {
    if(batch == null)
      batch = context.getConfiguration().get("batch","none");
    return batch;
  }

  @Override
  public String generatePartitionedPath(JobContext context,
   String topic,int brokerId, int partitionId,
   String encodedPartition) {
    return topic+"/"+encodedPartition;
  }

}
```

该Partitioner类在camus.properties文件中进行设置，它重载了默认行为：

```
elt.partitioner.class=wiley.streaming.camus.BatchPartitioner
```

6.5.1.4 在ETL过程中使用Hadoop

在将数据例行摄取到Hadoop中存储起来之后，也可以将其集成到ETL过程。本书主要关注实时分析，因此我们对此只作简要讨论。

对于Hadoop中的ETL流水线有许多选择。其中比较流行的是Pig和Hive。前者是Hadoop的脚本语言，由Yahoo！开发，可以用于ETL处理。后者是Hadoop的Map-Reduce框架的类SQL接口，它是数据库开发人员的主流选择。

这些工具以及其他一些工具，通常用于多步的流水线，用来生成一系列聚集计算结果。然后这些结果就可以为其他流水线或处理工具所使用。

同样，在这里仍然流行使用Hive，因为它可以与外部的查询工具相集成。此外，还有其他专门的商用工具可以直接与Hadoop一起使用。

也可以将数据从Hadoop中载入其他数据库环境。许多数据库供应商提供了从Hadoop到其数据库的连接器，用来简化该载入过程。另外一个更通用的选择是Sqoop。这是一个Apache项目，用于在Hadoop和其他数据存储之间进行批量传输。该项目的软件包中包含一个服务器，负责管理用于批量输入和输出的Hadoop作业。它由单独的客户端进程控制。

6.5.2　Lambda架构

本章讨论的所有存储系统都具有一个关键特性，那就是能够对更新机制进行调优。实际上，这意味着为了提高写速度而放弃事务特性。再加上数据流程机制的至少一次属性，这可能导致聚集结果的减少计数和重复计数，以及其他相关问题。

为了克服这个问题，Nathan Marz引入了Lambda架构的概念。在该架构中，实时系统像以前一样更新其数据存储，而不考虑事务。该架构接受这样一个概念，这些数据只是真实值的近似。

最终值是用数据仓库系统计算出来的。在多数例子中，这些最终值由大型Hadoop批处理作业来计算，然后由前端界面对选择哪个系统进行管理。另一种方法是使用同一个数据库来存储这两个系统的数据，并且一旦得到批处理系统的值，就用它们覆盖实时值。这种方法需要的资源较少，并且易于在前端管理，但实现起来比较难。对于实时系统的短期存储，以

及保存批处理系统“最终”结果的长期存储，使用不同的存储技术往往也是可取的。

6.6　总结

本章介绍了几种广受欢迎的数据存储技术的基础知识。实质上，只要付出足够的努力，所有的存储都可以在给定的场景下良好运行，但对于特定的应用来说，某些存储还是要比其他合适。这往往难以抉择，最好的办法就是尝试一番并进行实验。刚开始时，可以将数据存储扩展到现有数据上，这样做通常就可以排除不少选项。

现在，数据已经流入数据处理系统，数据处理系统也已经将输出放入合适的地方，是时候建立应用了。在下一章中，我们将建立一个简单的仪表板环境，从而为数据建立展示界面。许多流数据项目都是以建立仪表板应用为起点的，因为用户可以通过这样的应用“感知到”数据，并与之交互。

第二部分　流分析与可视化

这一部分内容包括：

第7章　流度量的交付

第8章　精确的聚集计算和交付

第9章　流数据的统计近似

第10章　使用略图近似流数据

第11章　流数据的应用

第7章　流度量的交付

对流数据进行采集、处理和存储之后，就应该将其交付给终端用户。以前，流数据应用利用的是各种类型的定制“胖客户机”。后来随着基于Web的接口的引入，应用演化为基于applet（使用Java）方式，或者（往往是）基于插件方式。例如，多年前，Adobe就通过Flash提供了流接口，它使用的是它的服务器生命周期线（LifeCycle server line）。

对于现代交付环境来说，无论是基于applet还是基于插件的方式都没多少吸引力。首先，随着真正的移动计算的兴起，基于Web的交付根本无法使用这些方法。另外，在桌面环境下，考虑到安全和用电量方面的问题，现代浏览器也对这些方法的使用有严格限制，这使得使用它们越来越困难。

还好，Web浏览器标准的发展已经可以支持海量数据的应用，乃至有流数据连接的应用。对于流数据应用来说，最重要的标准是服务器端推送事件协议（Server-Side Events，SSE）和Web Socket框架。这两个标准在各种浏览器广泛采用，它们都允许Web页面接收流式更新。

本章并不会对Web应用开发做完整讨论，而只对这个话题给出一个大体介绍，重点聚焦于实时应用。我们介绍了Web应用后端结构的基础知识，并使其适应实时应用的特殊问题，这些问题在某些方面不同于传统Web应用的对应问题。

将数据交付到前端之后，数据渲染就成为应该关注的主要问题。现代

Web浏览器提供了一些高级的数据渲染技术，这些技术甚至取代了仅仅几年前才出现的基于Flash的方法。这些技术相当完美，因为它们能够应用到很多种浏览器（甚至移动浏览器）中，并具有很高的渲染保真度。D3 JavaScript库是其中一个最流行的工具，它被广泛用于渲染高质量的可视化。本章会介绍D3工具包，以及D3的几个高层抽象（因为D3是一个相当底层的库）。

7.1 流Web应用

对传统的Web服务器来说，对连接的处理依赖于操作系统的多线程或多进程支持。在早期的Java Web服务器或公共网关接口（Common Gateway Interface）应用中，从Web浏览器建立新的连接时，就会派生新的操作系统级线程或进程（取决于操作系统和具体的服务器）来负责处理。

操作系统进程具有大量的内核开销。内核线程尽管要比进程轻量一些，但也会引入一些开销以及上下文切换代价。该模型的优势是，除了要注意共享资源，不需要程序员进行任何外部干预。缺点是，如果并发连接太多，需要的内核资源就无法忽略。实际上，只需要让操作系统派生过多的线程，就可以使它无法被访问。

为了绕过这个问题，Web服务器开始使用“非阻塞”I/O，而不是传统的线程模型。在该模型中，服务器启动相对少数的worker（通常每个系统内核一个worker），然后使用非阻塞I/O机制来处理连接。刚开始时，这些非阻塞I/O只是用于实际的Web连接，后来逐步扩展成为某些环境中的标准交互机制。

其中一个这样的环境就是node.js（常简称为Node），这是一个服务器端的JavaScript应用。它完全基于非阻塞的I/O库（具体的库取决于能从操作系统得到什么），使用单个运行线程。Node.js已经成为Web应用开发的主流框架，并且由于具备轻松处理大量持久连接的能力，它也成为构建流应用的明智选择。Node.js基于Google Chrome浏览器的V8引擎，V8这

个JavaScript引擎速度非常快，但它的效率不如基于Java的服务器。不过，如果在前端和后端使用JavaScript，就可以支持一些有趣的功能，对许多应用来说，这比性能更重要。

在本章剩下的内容中，我们将使用node.js和相关的库来构建流应用。本章假定你基本熟悉JavaScript开发，但并不特别要求熟悉node.js。

7.1.1 使用Node

尽管Node也是基于JavaScript的，Web应用开发中浏览器一方的程序员往往也会被Node的处理模型弄得晕头转向。其他Web应用框架（例如Ruby on Rails）的工程师也有同样的感受。这种困扰来自于Node的事件驱动回调风格，这种编程风格与多数程序员习惯的风格迥然不同。有必要在开头对该风格做个简要介绍，来帮助你理解Node是怎样工作的。

注意　Node项目进展很快，它会定期发布新版本。其中一些版本会引入或删除某些功能，这会导致版本之间不易兼容。为此，本书假定Node版本接近于0.10应用编程接口（API）。

7.1.1.1 回调驱动编程

在Java或Python等多数语言中，都假设操作是阻塞的。因此，当代码从文件中请求获取一行数据时，它除了等待什么都不做，直到完整获得这一行数据，才将其返回给程序。例如，下列Java代码从名为stream的InputStream中读取文件：

```
InputStreamReader reader = new InputStreamReader(stream);
BufferedReader    lines  = new BufferedReader(reader);

while(lines.ready()) {
  var line = lines.readLine();
  //Do something with the line here
}
```

在Node中，由于假定I/O是非阻塞的，上述编程风格通常不可行。（也有特例，但非常罕见。）取而代之的是，Node接口或者获得对象，或者响应事件。例如，下列代码中打开了一个文件，却不像Java程序那样设置一个函数来读取下一行，而是将函数绑定到处理数据行的line事件。我们没有在这个例子中显示出其他事件，它们也可以绑定到函数中，从而使程序对文件末尾或读进程产生的错误做出响应：

```
var readline = require('readline');
var lines    = readline.createInterface({
  input: stream
});
lines.on('line',function(line) {
  //Do something with line here
});
```

7.1.1.2 避免“回调金字塔诅咒”

当只有一个事件集要处理，而这些事件不需要被再次调用时，回调风格的编程很容易阅读和理解。当然，对于稍复杂一点的软件，这种情况不太可能出现，此时回调模型就不太好用了。例如，在下面的例子中，顺序操作就难以阅读和理解：

```
first(input,function(error,data) {
  second(data,function(error,moreData) {
    third(data,moreData,function(error,evenMoreData) {
      //Do something with the data...
      console.log("data: "+data+", moreData:"
        +moreData+" evenMoreData:"+evenMoreData);
    });
  })
});
```

这种情况被通俗地称为“回调金字塔（诅咒）”，常常会对前端和后端的JavaScript程序员造成困扰。有一些文体方面的方法，例如可以通过将操作重写为独立的命名函数来重新组织回调金字塔：

```
function doThird(data,moreData) {
  third(data,moreData,function(error,evenMoreData) {
  });
});

function doSecond(data) {
  second(data,function(error,moreData) {
    doThird(data,moreData);
  });
}

first(a,function(error,data) {
  doSecond(data);
});
```

但在现实中，要保持这种编程风格实际上非常麻烦。另一个方法是使用events库（该库是Node核心库的一部分），通过EventEmitter来协调顺序事件：

```
var coord = new require('events').EventEmitter();

coord.on('first',function(a) {
  first(input,function(error,data) { emit('second',data); }
});

coord.on('second',function(data) {
  second(data,function(error,moreData) {
    coord.emit('third',data,moreData);
```

```
    }
  });

  coord.on('third',function(data,moreData) {
    third(data,moreData,function(error,evenMoreData) {
    });
  });
```

这个例子与上一个例子非常相似，但是这里调用之间的相似性也暗示我们，可以创建一个能很容易地将调用串联起来的库。实际上，在许多实现了各种调用协调技术的库中，这已经得到了实现。其中一个名为async的此类库在本书中会大量用到，使用它可以将前面的例子实现为waterfall：

```
var async = require('async');
var input = ...;

async.waterfall([
  function(cb) {
    first(input,function(error,data) {
      cb(data);
    });
  },

  function(data,cb) {
    second(data,function(error,moreData) {
      cb(data,moreData);
    });
  },

  function(data,moreData,cb) {
    third(data,moreData,function(error,evenMoreData) {
      //Do something here
      cb(); //All done
    });
  }
]);
```

7.1.2　用NPM管理Node项目

为了管理Node项目代码，包括管理上一节使用的async包等依赖，

Node在node的安装包中为多数操作系统提供了npm（node包管理器，node package manager）。该工具提供的服务与Java环境的Maven以及Ruby环境的gem很相似。

7.1.2.1 启动Node项目

启动Node项目很简单，就是创建一个新目录，然后运行npm init：

```
$ mkdir myproject
$ cd project
$ npm init
```

开始时，NPM工具会询问一堆关于项目的问题。该过程结束时，会在npm的运行目录创建package.json文件：

```
This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (myproject)
version: (0.0.0) 1.0.0
description: My first node.js project
entry point: (index.js)
test command: mocha
git repository:
keywords:
author: Byron Ellis
license: (BSD)
About to write to /Users/bellis/Projects/wiley/chapter7/myproject
/package.json:
```

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "My first node.js project",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
  "repository": "",
  "author": "Byron Ellis",
  "license": "BSD"
}

Is this ok? (yes) yes
```

在用NPM管理的项目中包含README.md文件是约定俗成的做法。如果该文件不存在，NPM会定期提醒，直到该文件被加入。创建package.json文件之后，就可以通过加入dependencies部分来添加async这样的依赖：

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "My first node.js project",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
  "repository": "",
  "author": "Byron Ellis",
  "license": "BSD",
  "dependencies":{
    "async":"*"
  }
}
```

如果要安装依赖，只需要运行npm install命令来获取最新版本（这是因为上述代码中指定了“*”，而不是具体的版本号）：

```
$ npm install
npm WARN package.json myproject@1.0.0 No README.md file found!
npm http GET https://registry.npmjs.org/async
npm http 304 https://registry.npmjs.org/async
async@0.2.9 node_modules/async
```

另一种方式是用npm命令，并以save选项来安装async包：

```
$npm install async --save
```

这样会用与感兴趣模块对应的条目自动更新package.json的dependencies部分。

开发工作启动之后，最好的做法通常是将所有依赖的版本固定下来，以避免由于包的变化而引入的错误。NPM提供名为shrinkwrap的工具，该工具可以用来指定软件包的具体版本，其效果类似于Ruby gem中的Gemfile.lock文件。

现在，一切就绪，可以基于node.js开发Web应用了。

7.1.2.2 向项目中加入模块

任何重要的项目都应该将代码分解为易管理的模块，以提高可维护性。在Node中，模块实现为简单JavaScript文件，模块的导入是通过require语句来完成的。与某些语言（例如Java）不同，Node并不需要将模块放到特定目录，不过通常最好将其放到一个子目录中，这样可以把它们与运行服务器和其他应用的主文件区分开。

模块的工作方式有两种：只导出特定函数，或者将整个模块导出为函数。某些模块，例如本章后面用的connect模块，会同时用到这两种方法。要从模块中导出函数，只需要显式地将该函数加入模块中的exports

块：

```
function notExported() {
  console.log("This function can only be used internally");
}

exports.exported = function() {
  console.log("This can be called externally");
}

exports.callInternal = function() {
  notExported();
}
```

如果这些函数位于lib/module.js文件中，那就可以通过require函数导入和使用：

```
var mod = require('./lib/module.js');

mod.exported();
// Prints "This can be called externally"

mod.callInternal();
// Prints "This function can only be used internally"

mod.notExported();
// Throws an exception "Object #<Object> has no method 'notExported'"
```

要返回一个模块级的函数，需要为module.exports赋予一个函数，以覆盖exports对象：

```
exports = module.exports = function() {
  //Do something here
};
```

由于函数是对象，可以用以上方式继续加入其他exports。

7.1.3　基于Node开发Web应用

Node默认带有用来创建和使用Web应用的API。它包含一个底层的http（或https）库，可以用来编写非常简单的Web服务器。对于像查询字符串解析、数据编码和解码等常见的Web服务器函数，Node甚至有内置的API。

要实现一个基本的服务器，只需要几行代码：

```
var http         = require('http')
,   url          = require('url')
,   querystring = require('querystring')
;

http.createServer(function(request,response) {
  var query = querystring.parse(url.parse(request.url).query || "");
  response.writeHead(200,{'content-type':"text/plain"});
  response.end("Hello "+(query.name || "Anonymous")+"\n");
}).listen(3000);
```

当然，这是非常底层的方式，一点也没有利用现代Web框架所提供的便利。要提供Web框架这样的高层接口，你有多种选择。其中一个最流行的方法就是connect框架搭配express中间件。这两个包构成了本章开发Web应用的基础。

7.1.3.1 HTTP中间件与connect框架

connect框架常被拿来与Ruby的rack项目相比。它建立在基本的HTTP服务器库之上，提供称为中间件（middleware）的可链接执行的代码段。这些代码段（核心库中包含二十几个）用来提供Web服务器的各种组件，例如查询字符串的解析、认证或文件服务。

要实现能够传输静态文件的基本服务器，只需要下列几行代码：

```
var connect = require('connect')
;

var app = connect()
.use(connect.static('public'))
.use(function(request,response) {
  response.writeHead(404,{'content-type':'text/plain'});
  response.end("File Not Found");
})
.listen(3000);
```

该服务器首先通过调用conncet（）创建一个connect应用，然后用use（）函数将中间件链接在一起。中间件是按照它被加入每个请求的顺序来执行的。

编写定制的中间件也很容易，因为中间件就是由函数组成的。所有的中间件函数都是function（request，response，next）这样的形式，函数可以修改request和response的值，之后可以选择性地调用next函数以进入下一个中间件。

中间件的配置通常是在以中间件自己作为返回值的函数中进行的。这使得中间件可以利用JavaScript的词法作用域进行配置。例如，要实现一个简单版本的static中间件，可以使用如下代码：

```
var url      = require('url')
,   path     = require('path')
,   fs       = require('fs')
;

function static(srcDir) {
  return function(request,response,next) {
    if(request.method != "GET") return next();
    var p = path.join(process.cwd(),
      srcDir,url.parse(request.url).pathname);
    fs.exists(p,function(yes) {
      if(!yes) return next();
      fs.createReadStream(p).pipe(response);
    });

  }
}
```

在上面的例子中，static函数返回另一个函数，后者是在srcDir中的源目录配置的。这使得我们可以通过调用use（static（“public”））,在connect中使用该static函数。包含在其他的函数中的函数称为“内部函数”。利用JavaScript的词法作用域特性，内部函数可以访问“外部”函数中的变量。当请求沿着中间件链前行时，内部函数首先会检查所请求的方法是否是GET，如果不是，那就调用next函数进入下一个中间件；然后，构造指向所请求文件的路径，如果在该路径中找到了文件，那就用Node ReadStream类的pipe方法将其通过管道发送给HTTP响应。注意，这时没有调用next函数。这是因为已经向客户端返回了响应，故而此时应该终止请求链。

7.1.3.2　express.js Web框架

express框架致力于构建成熟的Web应用，它建立在connect中间件的基础上，正如Ruby语言中的Sinatra或Rails建立在Rack之上。Express的设计目标是支持模型-视图-控制（Model-View-Control，MVC）模式的Web

应用，这种模式很适合流数据应用。使用express.js的方式与启动conncet应用很相似：

```
var express = require('express')
,   path    = require('path')
,   app     = express()
;
app
.set("port",argv.port)
.set("views",path.join(__dirname,"views"))
.set("view engine","jade")
.use(express.favicon())
.use(express.logger('dev'))
.use(express.json())

.use(express.urlencoded())
.use(express.methodOverride())
.use(app.router) //MVC routing
.use(require('less-middleware')({
  src: path.join(__dirname, 'public')
}))
.use(express.static(path.join(__dirname, 'public')));


app
.listen(argv.port);
```

与前面的connect例子类似，首先初始化应用，然后加入基础的中间件。多数情况下会用express中间件取代connect中间件，但第三方的connect中间件仍然可以与express一起使用。这里，默认中间件负责对URL各个部分以及网页正文进行解析，还负责传递public目录的静态文件，并用less软件包渲染层叠样式表（Cascading Style Sheets，CSS）。less软件包与其他一些软件包一起，被Twitter的Bootstrap用来简化应用的样式渲染。“Jade”视图引擎提供了模板语言，用来创建应用中的动态Web页面。在下一节中，使用该引擎来描述仪表板应用的布局，以使数据流可以流入仪表板应用。

7.1.4 基本的流仪表板

本节使用express构建了一个流仪表板的框架，这大概是第一个也是最常见的流数据应用。本节给出了该仪表板的框架和渲染方法，下一节将流数据的后端与前端联系起来。

7.1.4.1 Bootstrap布局

该仪表板使用的布局基于Twitter Bootstrap 3框架。这是一个轻量级的框架，能与jQuery完美集成，并且很容易集成移动特性。由于仪表板的所有部分都会用到Bootstrap，因此最好把它直接加入整个Web应用的布局中。默认的express布局名为layout.jade，它位于views目录。要包含Bootstrap，应该加入对应的样式表和脚本链接，参见下列代码中的加粗部分：

```
doctype 5
html
  head
    title= title
    meta(name='viewport',content='width=device-width,
      initial-scale=1.0')
    link(rel='stylesheet',href='/stylesheets/bootstrap.min.css')
    link(rel='styleshhet',
      href='/stylesheets/bootstrap-theme.min.css')
    block styles
  body
    block content
    script(src='/javascripts/jquery.min.js')
    script(src='/javascripts/bootstrap.min.js')
    block scripts
```

除了常见的内容块，该布局还要包含专门的块，用来自定义样式，以及为特殊视图加入专门的脚本。

7.1.4.2　仪表板视图

该仪表板应用有一个索引视图，该视图渲染了要显示在仪表板上的所有度量。典型的仪表板由各种小面板组成，小面板上是各种度量或可视化。我们在7.2节讨论数据的可视化。现在，第一个任务是构造度量面板。Bootstrap已经包含构建简单面板的样式和标记（markup）。你可以用下列HTML片段作为起点来构建新面板：

```
<div id="first-metric" class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">Header</h3>
  </div>
  <div class="panel-body">
    <!-- Body Goes Here -->
  </div>
  <div class="panel-footer">My First Metric</div>
</div>
```

该标记作为Jade模板语言的mixin模板。通过使用mixin特性，可以很容易地为大量度量快速生成布局。mixin为度量定义了所有的基本数据字段，因此在后端将数据与字段绑定就容易很多。该模板语言还支持条件语句，因此标题和脚注都可以是面板的可选元素：

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      h1(data-counter="#{id}")= "--"
      h3
        span.glyphicon(data-direction="#{id}")
        span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

然后将mixin加入普通的Bootstrap布局中，从而用sindex.jade视图（位于views director）中的若干不同度量来构建简单的仪表板：

```
block styles
  link(rel='stylesheet',href='/stylesheets/dashboard.css')

block content
  div(class="container")
    div.page-header: h1 An Important Dashboard
    div(class="row")
      div(class="col-lg-3")
        +metric("first","Metric 1","My First Metric")
      div(class="col-lg-3")
        +metric("second","Metric 2","Another Metric")
      div(class="col-lg-6")
        +metric("third","Metric 3","A Bigger One")
    div(class="row")
      div(class="col-lg-2")
        +metric("m4")
      div(class="col-lg-2")
        +metric("m5")

      div(class="col-lg-2")
        +metric("m6")
      div(class="col-lg-2")
        +metric("m7")
      div(class="col-lg-2")
        +metric("m8")
      div(class="col-lg-2")
        +metric("m9")
```

上述代码片段构建了一个带有简单标题的两行仪表板。第一行有3个宽度不一的面板，第二行有6个等宽面板。在浏览器中渲染后，该仪表板大致应该如图7-1所示。

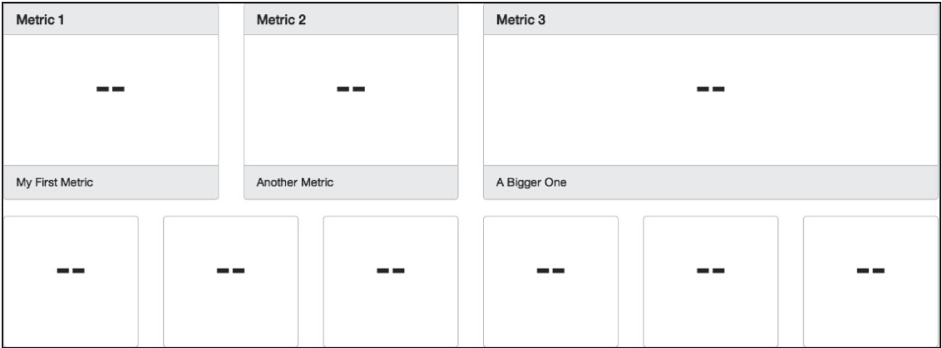


图　7-1

7.1.4.3 更新仪表板

这里要渲染的度量都是简单的计数器。本章后面会介绍更复杂的图形化度量，第8章会以时间序列展示为例介绍聚集计算度量。

为了更新仪表板上的计数器，必须将计数器注册到应用中，这样它们才能开始接收数据。这是通过使用所谓的数据属性来完成的。自定义数据属性是HTML5的发明，它支持将任意的用户属性与文档对象模型（Document Object Model，DOM）的各种元素绑定。

jQuery和其他框架都提供了选择器，用来查询特定数据元素是否存在。要找到仪表板中所有的data-counter元素，查询应该如下所示：

```
$('*[data-counter]').each(function(i,elt) {
    //elt is the document element
  });
```

由于上一节提到的mixin指定了data-counter字段中的度量，可以用该查询将计数器绑定到自定义事件，这样只要触发事件，就会更新计数器。首先，选择要绑定的每个计数器：

```
function bindCounters() {
  $('*[data-counter]').each(function(i,elt) {
    bindCounter($(elt).attr("data-counter"),$(elt))
  });
}
```

接下来找到方向标志，修改计数器度量中包含的数量字段，再将这些与data：＜counter name＞形式的自定义事件绑定。该实现位于public/javascripts/dashboard.js，如下所示：

```
function bindCounter(counterName,$elt) {

  $elt.text("--");

  var $dir = $('span[data-direction='+counterName+']');
  var $chg = $('span[data-change='+counterName+']');

  var last = NaN;

  $(document).on('data:'+counterName,function(event,data) {
    $elt.text(data);
    if(isFinite(last)) {

      var diff = data - last;
      if(diff > 0) {
        $dir.removeClass("glyphicon-arrow-down");
        $dir.addClass("glyphicon-arrow-up");
      } else {
        $dir.removeClass("glyphicon-arrow-up");
        $dir.addClass("glyphicon-arrow-down");
      }
      diff = Math.abs(diff);

      if($chg.hasClass("percentage")) {
        diff = Math.round(1000*diff/last)/10;
        $chg.text(diff+"%");
      } else {
        $chg.text(diff);
      }

    }
    last = data;
  });

  $chg.on('click',function() { $chg.toggleClass("percentage"); });
}
```

上述代码将需要用来更新计数器以及方向标志的所有信息都包装起来。它还将一个click事件与方向标志绑定，从而支持在完全显示和百分比显示两种方式间切换。

为了看一下这些度量的实际效果，下面给出了一个简单的驱动程序，用来以随机值触发各计数器事件。我们将该触发器设置为在1到5秒的范围随机触发。这段代码位于public/javascript/counter.js：

```
$(document).ready(function() {
  $('*[data-counter]').each(function(i,elt) {
    var name = $(elt).attr("data-counter");
    setInterval(function() {
      $(document).trigger("data:"+name,
        Math.round(10000*Math.random()));
    },1000+Math.round(4000*Math.random()));
  });
});
```

如图7-2所示，这些计数器现在都准备就绪，可以接收从服务器流入的数据。

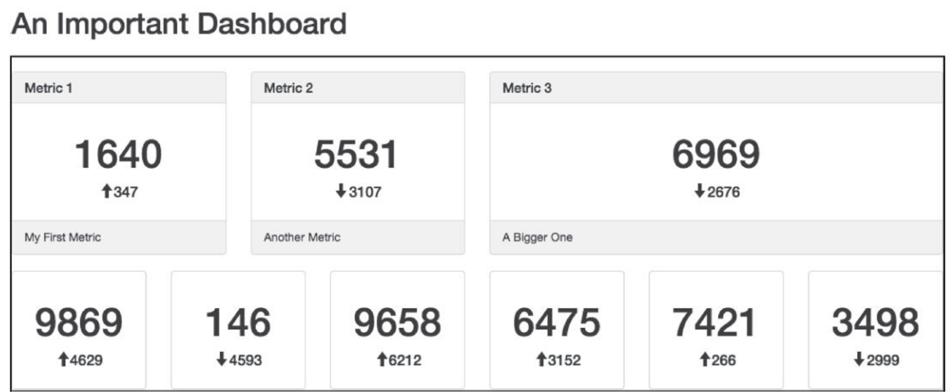


图　7-2

7.1.5　向Web应用加入流

仪表板中的计数器准备好接收数据流之后，对于基于Web的数据交付来说，有两种通信协议可以选择：SSE协议和WebSocket协议。这两种协议在包括移动浏览器在内的现代浏览器中都得到了广泛支持。两者各有优缺点，因此选择其中哪一种取决于偏好，而不是具体需求。

本节介绍怎样使用这两种协议来与后端服务器通信。这两种协议互不干扰，因此后端服务器如果都支持，那最好不过了。

7.1.5.1　服务器端计数器的管理

在介绍通信协议之前，必须在服务器端维护仪表板中使用的计数器。否则，客户端连接服务器之后，根本就没有东西发送给客户端。在下面的

示例应用中，计数器是在一个单例对象中维护的，该对象派生自Node的内置类EventEmitter，其实现位于dashboard/lib/counters/index.js：

```
var events = require('events')
,   util   = require('util')
;

function Counter() {
  //Returns the singleton instance if it exists
  if(arguments.callee.__instance) {
    return arguments.callee.__instance;
  }
  arguments.callee.__instance = this;

  this._state = { };

  events.EventEmitter.call(this);
}
util.inherits(Counter,events.EventEmitter);
module.exports = new Counter();
```

该单例对象存储于arguments.callee._instance变量。这使得我们可以用require语句导入计数器对象。

这里，计数器对象的状态存储在对象自身中。比较复杂的仪表板应该将该数据持久化，以保证重启后数据不丢失。我们会在本节稍后的内容中讨论该问题。

计数器状态的更新是通过调用计数器对象的方法进行的。这里支持的基本操作是get、set和increment方法：

```
Counter.prototype.state = function(cb) {
  return cb(null,this._state);
};

Counter.prototype.get = function(counter,cb) {
  return cb(null,this._state[counter] || 0);
};

Counter.prototype.set = function(counter,value,cb) {
  this._state[counter] = (value || 0);
  this.emit("updated",counter,this._state[counter]);
  return cb(null,this._state[counter]);
};

Counter.prototype.increment = function(counter,amount,cb) {
  var self = this;
  this.get(counter,function(err,count) {
    self.set(counter,count + 1*(amount||1),cb);
  })
};
```

注意，只要计数器有变化，上述代码中的set方法都会发送一个updated事件。这是流链接与计数器的主要交互活动。该方法很简单，就是在链接处于打开状态时监听该事件，并在链接关闭时退出监听。

既然我们已经可以在服务器端维护计数器状态，那就需要一个简单的API来支持状态的更新。在后面两节我们会分别通过SSE和WebSocket将该API附加到客户端接口。通过express很容易为计数器单例对象实现简单的REST风格API，这部分代码位于dashboard/index.js，如下所示：

```
var counters = require('./lib/counters');
app.set("counters",counters);


app.get('/api/v1/incr/:counter', function(req,res) {
  counters.increment(req.params.counter,req.query.amount,
    function() {
      res.end("OK\n");
    });
});
app.get('/api/v1/counter/:counter',function(req,res) {
  counters.get(req.params.counter,function(err,data) {
    res.end(data+"\n");
  });
});

app.get('/api/v1/state',function(req,res) {
  counters.state(function(err,data) {
    for(var counter in data) {
      res.write(counter+": "+data[counter]+"\n");
    }
    res.end();
  });
});
```

7.1.5.2　服务器端推送事件协议

服务器端推送事件协议（Server Sent Events，SSE）是由Web超文本应用技术工作组（Web Hypertext Application Technology Working Group，WHATWG）加入HTML5规范的。SSE协议最早于2006年提出，它使用HTTP作为传输机制，内容类型（Content Type）为text/event-stream。基本的响应是前缀为“data：”并以\n\n（两个换行符）结尾的一行。如果要让响应变成多行，那就以“data：”为前缀但只使用一个换行符。Web服务器可以分别使用“retry：”、“event：”和“id：”来设置可选的重试超时时间、事件名称和ID字段。一个完整的响应可以是如下形式：

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/event-stream
Cache-Control: no-cache
Connection: keepalive
Transfer-Encoding: chunked

event:alpha
id:1234
retry:10000
data:{"hello":"world"}

id:12345
data:{"again":"and again"}
```

这个响应返回一个简单的单行消息，消息的id为1234。我们还将重试时间从30秒改为10秒。要读取这条消息，支持SSE的浏览器会使用EventSource类。这个类会触发与响应中的事件名称相对应的事件。而如果响应中没有指定事件，那就触发message事件。

要修改要响应的仪表板应用示例，服务器就要触发计数器事件来传输一个JSON对象，该对象包含对各计数器的更新。

首先，应用必须建立与服务器的连接。这通常是由EventSource对象来完成的，该对象的输入参数为一个指向SSE连接的URL。在仪表板应用实例中连接位于/api/v1/dashboard，实现位于dashboard/public/javascripts/sse.js：

```
$(document).ready(function() {

  var events = new EventSource('/api/v1/dashboard');
```

EventSource对象会发送一系列事件，其中包括前面介绍的消息事件。这些事件往往会给出有用的调试信息。是否处理这些事件是可选择的，其中最重要的事件是open和error事件，前者在成功建立事件流时被

调用，后者在事件流被关闭或发生另外的错误事件时（例如500号错误事件）被调用。在下面这个简单例子中，这些事件只是简单地向控制台报告自己已经被调用，用来帮助调试：

```
events.addEventListener('open',function(e) {
  if(window.console) console.log("opened SSE connection");
});

events.addEventListener('error',function(e) {
  if(e.readyState != EventSource.CLOSED) {
    if(window.console) console.log("SSE error");
  } else
    if(window.console) console.log("closed SSE connection");
});
```

最后，对counters事件进行处理，将计数器状态分发给已经建好的listener：

```
  events.addEventListener('counters',function(e) {
    var data =  JSON.parse(e.data);
    for(counter in data)
      $(document).trigger('data:'+counter,data[counter]);
  });


});
```

为了响应该事件，express应用使用SSE中间件函数来提供SSE协议，定义了与用来创建EventSource对象的端点（endpoint）相对应的路由（route）：

```
app.get('/api/v1/dashboard', sse(), routes.dashboard);
```

SSE中间件是一段非常简单的代码，它的实现位于dashboard/lib/sse/index.js中。该实现负责维护每个事件的唯一消息ID，并支持默认事件名：

```
module.exports = function(options) {
  options = options || {};

  return function sse(req,res,next) {

    res._counts = {};
    res._nextId = function(type) {
      type = type || "message";
      count = (this._counts[type] || 0) + 1;
      this._counts[type] = count;
      return count;
    }
```

当连接开始时，必须检查调用的来源是否支持SSE。如果是，就发送响应的内容类型和其他参数，以开始连接：

```
if(req.accepts("text/event-stream")) {
      req.setTimeout(Infinity);
      res.writeHead(200,{
        'Content-Type':  'text/event-stream',
        'Cache-Control': 'no-cache',
        'Connection':    'keepalive'
      });
    } else {

    res.writeHead(200,{
      'Content-Type': 'application/json',
      'Cache-Control': 'no-cache',
      'Connection': 'keepalive'
    })
  }
```

继续进行之前，中间件要向响应对象中加入函数，从而以合适的格式传输SSE事件：

```
    res.event = function(event,type,opt) {
        opt   = opt   || {};
        type = type || options.event;

        if(type)
          this.write("event:"+type+"\n");
        this.write("id:"+this._nextId(type)+"\n");
        if(opt.retry)
          this.write("retry:"+opt.retry+"\n");
        this.write("data:"+event+"\n\n");
      }

      res.json = function(json,type,opt) {
        this.event(JSON.stringify(json),type,opt);
      }

      next();

    }

  };
```

最后，端点在连接开始时接收当前计数器状态，并立即发送计数器数据。然后，该连接的处理函数（handler）将自己附加到Counter对象，以从服务器获取计数器未来的更新，获得更新之后，再将更新传递给客户端。处理函数的实现如下所示，这部分实现代码位于dashboard/routes/index.js：

```
exports.dashboard = function(req,res) {
  var counters = req.app.get('counters');

  counters.state(function(err,data) {
    console.log(data);
    res.json(data,"counters");
  });

  function update(counter,value) {
    var msg = {};msg[counter] = value;
    res.json(msg,"counters");
  }

  counters.on("updated",update);

  res.on("close",function() {
    counters.removeListener("updated",update);
  });

};
```

现在一切就绪，如果你在命令行中敲入状态更新，仪表板应该可以作出响应。你可以尝试启动仪表板，然后在命令行窗口运行下列命令：

```
$ curl http://localhost:3000/api/v1/incr/first?amount=25
OK
$ curl http://localhost:3000/api/v1/incr/second?amount=25
OK
$ curl http://localhost:3000/api/v1/incr/third?amount=50
OK
```

上述命令运行之后，仪表板的显示应如图7-3所示。

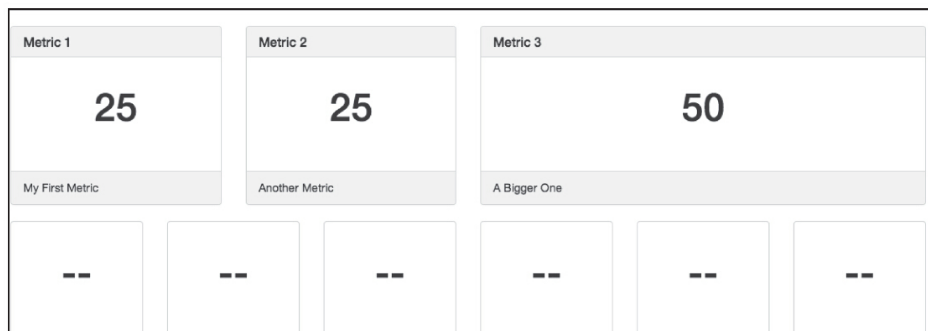


图 7-3

Safari和Chrome等基于Webkit内核的浏览器都支持SSE。它们在iOS和Android系统的移动端对应版本也是一样。Firefox和Opera浏览器同样支持SSE，但Internet Explore或默认的安卓浏览器（由于某些原因并不是Chrome）则不支持。不过，由于SSE规范只依赖于基本的HTTP，因此我们可以使用“polyfill”库来提供支持。截止到撰写本书的时间，对SSE存在3个著名的polyfill库，其中Remy Sharp的polyfill库（http://github.com/remy/polyfills/ ）提供的接口与原始实现的接口匹配得最好，并且可以被Internet Explorer 7及后续版本所支持。

7.1.5.3 WebSocket协议

WebSocket协议又名RFC 6455，是针对“胖”客户端的双向通信层标准。它在80端口维护通信，但是与SSE不同，它将连接“升级”为通过握手过程来支持通信。

自2010年以来，该标准已经有了多个不同的版本，并且其现代安全版得到了浏览器的广泛支持。Internet Explorer 10、Chrome 14、Safari 6、

Firefox 6和Opera的现代版本及后续版本都支持WebSocket。这些浏览器的移动版也是如此。Chrome还支持该协议的试验性SPDY版本，但到撰写本书的时间为止，还必须用启动开关来启用该功能，因此实际上该版本尚未真正投入使用。

由于WebSocket协议比SSE复杂，因此仪表板应用不是直接实现服务器和客户端，而是依靠一个库来支持WebSocket。对于Node存在多个WebSocket库，其中最流行的一个是socket.io库，它既提供了服务器端库也提供了客户端库，并且这两个库具有相同的接口。它还将浏览器的API差异标准化，从而提供了一个建立在所有平台之上的公共接口。使用socket.io最简单的方式是，将其服务器接口直接附加到Express应用，并把它作为应用的单独接口来看待。如果要这样做，需要创建一个http.Server对象，以替换应用的listen（ ）方法。在仪表板例子中，这需要在应用程序初始化时创建server对象，这部分代码位于dashboard/index.js：

```
var express  = require('express')
,   routes   = require("./routes")
,   path     = require('path')
,   app      = express()
,   server   = require('http').createServer(app)
,   sse      = require('./lib/sse')
;
```

接着用server.listen（argv.port）替换app.listen（argv.port），并在应用程序初始化后将socket.io的代码附加到服务器中：

```
server.listen(argv.port);
var ws = require('socket.io').listen(server);
ws.sockets.on('connection',function(socket) {
  function update(counter,value) {
    var msg = {};msg[counter] = value;
    socket.emit('counters',msg);
  }
  socket.emit('counters',counters.state);
  counters.on('updated',update);
  socket.on('disconnect',function() {
    counters.removeListener('counters',update);
  });
});
```

该实现本身与SSE版本中的dashboard端点很相似。当计数器被修改时，这个变化会被传递给仪表板，从而更新界面。

客户端的实现也与SSE版很相似。首先，必须包含socket.io库。由于该库自身提供了服务器和客户端版本，因此当服务器被附加到Express服务器时，它会在/socket.io/socket.io.js自动创建一个端点，来提供相应的客户端库。然后只需要简单修改仪表板的HTML来支持WebSocket，修改内容包括下列额外的脚本标记：

```
block scripts
  script(src='/socket.io/socket.io.js')
  script(src='/javascripts/dashboard.js')
  script(src='/javascripts/ws.js')
```

接下来只剩下一件事，像SSE版那样，当计数器的更新事件通过WebSocket连接到来时，将其重新发布。这里只需要很少的代码，因为事件已经通过socket.io库被转换回JSON格式：

```
$(document).ready(function() {
  var socket = io.connect("/");
  socket.on('counters',function(data) {
    for(counter in data)
      $(document).trigger('data:'+counter,data[counter]);
  });
});
```

提示　由于WebSocket和SSE使用不同的端点和不同的协议，服务器可以直接支持这两者，都作为流事件的传输机制。如果你想要支持大量设备甚至本地实现（在移动设备的情况下），这会特别有用。

注意　还存在支持服务器和浏览器之间通信的第三种技术，它就是WebRTC标准。该标准最初用于点到点的视/听通信，但它也具备通过网络传输任意数据的能力。目前支持该标准的浏览器很有限，只有Firefox和Chrome的新桌面版本。希望不远的将来这种情况能够有所改变，从而让我们有一个新的丰富的流数据接口。

7.1.5.4　基于Redis的仪表板

基于内存的计数器状态提供REST风格的接口，这在开发时很有用，前面几章讨论了实时流Web应用的数据流水线。此时，流处理是由Storm或Samza负责的，流的更新则被发送到某种持久存储中。

聚集计算和计数器的维护在第8章中有更详细的论述。很容易将前面的简单计数器仪表板改为基于持久存储，然后就可以由流处理环境来驱动。

下面的例子使用Redis，因为它是一个比较简单的键-值存储，并且也具有发布订阅功能，可以用来驱动更新。没有这项功能的持久存储就需要额外的消息功能来驱动仪表板服务器（或在服务器端使用轮询来驱动更新）。

在下面的示例仪表板中，服务器跟踪一个Redis哈希值，这个值可以通过外部接口或仪表板接口来更新。首先，要将Redis配置参数加入

dashboard/index.js中的服务器的参数中：

```
var argv    = require('optimist')
.usage("Usage: $0")
.options("port",{alias:"P",default:3000})
.options("redis",{alias:"r"})
.options("counters",{alias:"c",default:"counters"})
.argv
;
```

这些可以用来驱动对相应计数器库的选择：

```
var counters = argv.redis ?
  require('./lib/counters-redis')(
    argv.redis,
    {counters:argv.counters})
: require('./lib/counters');
```

接口代码已经以回调风格实现，因此这部分接口不需要修改。同样，已经实现了SSE和WebSocket接口来响应Counter类发送的事件。唯一需要实现的是新的Counter类。

新Counter类并不是简单地创建一个单例对象，而是要获得用来识别Redis服务器位置的参数。这些参数用来构建两个Redis连接。第一个连接称为client，用于所有的常规Redis操作。第二个连接称为pubsub，用来为Redis操作处理事件传递。这里使用了两个客户端连接，因为当使用subscribe命令时，Node的Redis客户端会处于特殊模式，该模式会阻止客户端执行常规操作。这部分代码位于dashboard/lib/counters-redis/index.js：

```
function Counter(host,options) {
  //Returns the singleton instance if it exists
```

```
    if(arguments.callee.__instance) {
      return arguments.callee.__instance;
    }
    arguments.callee.__instance = this;
    events.EventEmitter.call(this);

    this.options     = options || {};
    this.client      = redis.createClient(this.options.port || 6379
     ,host);
    this.counterName = this.options.counters || "counters";

    this.pubsub = redis.createClient(this.options.port || 6379,host);
    var self = this;
    this.pubsub.on('ready',function() {
      self.pubsub.subscribe(self.counterName);
    });

    this.pubsub.on('message',function(channel,message) {
      if(channel == self.counterName) {
        self._sendUpdate(message);
      }
    });

  }
  util.inherits(Counter,events.EventEmitter);
  module.exports = function(host,options) {
    new Counter(host,options);
  }
```

当新的计数器事件到达对与服务器监控的哈希值相对应的pubsub连接时，_sendUpdate方法就被用来通知所有的已订阅客户端：

```
Counter.prototype._sendUpdate = function(counter) {
  var self = this;
  this.get(counter,function(err,data) {
    if(err) return;
    self.emit("updated",counter,data);
  });
};
```

这使得外部处理系统可以通过Redis向相应channel发送一个publish事件，从而将更新推送给客户端。还使得set和increment事件进一步与Counter的实现解耦：

```
Counter.prototype.set = function(counter,value,cb) {
  var self = this;
  this.client.hset(this.counterName,counter,value,
    function(err,data) {
      if(err) return cb(err,data);
      self.client.publish(self.counterName,counter);
    });
};

Counter.prototype.increment = function(counter,amount,cb) {
  var self = this;
  this.client.hincrby(this.counterName,

    counter,amount,function(err,data) {
      if(err) return cb(err,data);
      self.client.publish(self.counterName,counter);
      cb(err,data);
  });
};
```

注意increment方法不再需要依靠get和set方法，因为Redis提供自己的原语来原子化地增加值。Get和set方法现在只需要查询Redis存储，这就解释了尽管并不严格要求，但还是在原来的实现中使用了回调结构：

```
Counter.prototype.state = function(cb) {
  this.client.hgetall(this.counterName,cb);
};

Counter.prototype.get = function(counter,cb) {
  this.client.hget(this.counterName,counter,cb);
};
```

再次启动仪表板之后，对于生成图7-3的计数器的设置命令，运行该命令应该会返回所需要的结果。如果要查看从仪表板外部更新的事件，现在可以从Redis命令行客户端推送事件：

```
$redis-cli
redis 127.0.0.1:6379> hset counters m4 4
(integer) 1
redis 127.0.0.1:6379> hset counters m5 5
(integer) 1
redis 127.0.0.1:6379> hset counters m6 6
(integer) 1
redis 127.0.0.1:6379> publish counters m4
(integer) 1
redis 127.0.0.1:6379> publish counters m5
(integer) 1
redis 127.0.0.1:6379> publish counters m6
(integer) 1
```

这时得到的仪表板如图7-4所示。

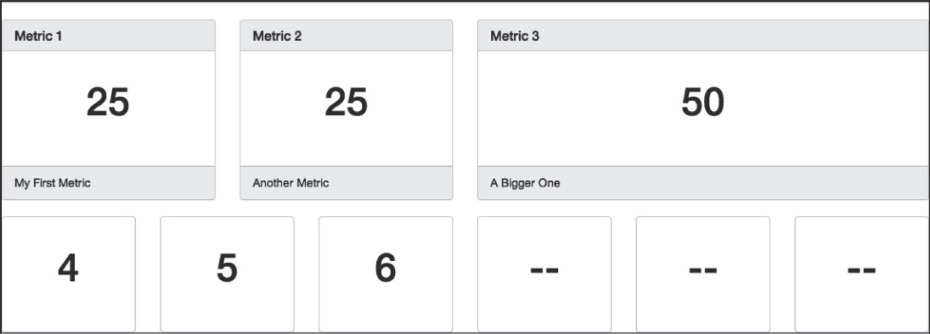


图　7-4

7.2 数据可视化

多年来，Web浏览器的数据可视化只有两种选择：由服务器生成静态图像和基于插件的可视化（通常用Flash实现）。

静态图像的灵活性不错，但对于数据的流式展示来说，这种可视化方法在技术上不可行。要达到流式展示的要求，至少要每30秒至1分钟更新1次。你可以在许多服务器监控解决方案（如Graphite）中见到该接口。这些方案在服务器端渲染静态图像，然后偶尔将其传输给客户端。

这种方法不是实时的，也不提供任何交互。另外，它也是一种重负荷的方法，因为一旦数据有微小改变，就需要重传整个图像。为了解决这个问题，常使用基于Flash或Java的插件，来支持增量式的数据更新。

当有必要支持旧式（甚至是劣质）浏览器时，这两种可视化方法仍然可以派上用场。但是，现代浏览器可以利用HTML Canvas和内联（Inline）SVG，以支持直接在浏览器中使用丰富的非插件可视化技术。

7.2.1 HTML5 Canvas和内联SVG

HTML Canvas元素和内联SVG不需要使用插件，就可以在HTML文档中提供丰富的渲染接口。顾名思义，Canvas元素提供了基于像素的接口，该接口类似于本地桌面或移动应用中使用的图层渲染接口。内联SVG则使用DOM来构建图形表示，并将图形表示与文档的其他部分一起渲染。

这两种方法各有利弊。Canvas接口渲染速度要快一些，但对于复杂渲染的原生支持相对较少。内联SVG内置了对于分层和分组的支持，并在许多浏览器中支持CSS样式表，但由于需要维护场景图，因此渲染大型场景时要慢很多。应该选择哪种技术通常取决于渲染性能，因此最常见的方式是使用内联SVG，直到性能出现问题再改用Canvas接口。

7.2.1.1　使用HTML5 Canvas

HTML Canvas元素是依赖分辨率的图层渲染技术。包括Internet Explorer（版本7和版本8需要外部库，版本9及以后版本直接支持canvas元素）在内的许多现代浏览器都支持Canvas。

要使用Canvas，需要向页面中加入一个元素。对mixin度量略作修改，在页面中的每个度量后面加入一个canvas元素，并用数据元素进行相应的标记，以供日后更新。该mixin类分散实现于dashboard/views目录下的各种.Jade视图文件中，例如dashboard/views/canvas_ex1.jade：

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        canvas(data-canvas-gauge="#{id}")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
          span.glyphicon(data-direction="#{id}")
          span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

如果不使用样式，元素就是顺次放置，这样下一步就是将元素逐个堆叠起来，并设置z-index值。设置z-index值是为了让元素按照正确的顺序

渲染。这段代码位于dashboard/public/stylesheets/dashboard.css：

```
.metric div.canvas-wrap {
  position: absolute;
  width: 100%;
  height: 100%;
  z-index: 1;
}

.metric div.canvas-wrap canvas {
  width: 100%;
  height: 100%;
}


.metric div.value-wrap {
  position: relative;
  width: 100%;
  z-index: 2;
}
```

要为每个计数器绘制仪表盘（gauge），首先需要获取canvas的上下文：

```
$('*[data-canvas-gauge]').each(function(i,elt) {
  fixup(elt)
  var name = $(elt).attr("data-canvas-gauge");
  var ctx  = elt.getContext("2d");
```

Fixup函数对canvas包装器的位置进行了略微修改，从而使得canvas位于其后面面板的中心位置。它还设置canvas图层的分辨率，使之于该元素的分辨率相匹配。该函数的实现位于dashboard/public/javascripts/canvas.js：

```
function fixup(elt) {
  var $elt = $(elt);
  var $wrap    = $elt.parent();
  var $parent = $wrap.parent();

  var pos      = $parent.position();

  $wrap.width($parent.width());
  $wrap.height($parent.height());

  elt.width = $parent.width();
  elt.height= $parent.height();

}
```

Canvas元素实际上有两个大小：元素本身的大小和canvas图层的大小。前者通过style属性设置，后者通过width和height属性设置。这容易让canvas新手感到困惑，为此在上述代码中，我们让canvas分辨率等于元素大小，以方便使用。

现在已经获取了图层上下文，canvas也已经移到正确的位置，可以开始绘制。元素的创建是用各种“单步动作”命令和曲线命令来进行的。前者的例子包括moveTo和lineTo命令，后者的例子包括arcTo命令。然后用stroke命令绘制轮廓，或者用fill命令填充。与多数类似的实现一样，图层也有方便的函数来绘制矩形。在这里，仪表盘是用drawGauge函数绘制的，这个函数实现于dashboard/public/javascripts/canvas.js，负责绘制图7-5中的圆形仪表盘：

```
function drawGauge(ctx,value,max) {
    //Clear the gauage
    ctx.fillStyle = "#fff";
    ctx.clearRect(0,0,ctx.canvas.width,ctx.canvas.height);

    //Draw the gauge background
    var centerX = (ctx.canvas.width) / 2;
    var centerY = (ctx.canvas.height) /2;
    var radius  = Math.min(centerX,centerY) - 12.25;

    ctx.beginPath();
    ctx.arc(centerX,centerY,radius,0.6*Math.PI,2.4*Math.PI)
    ctx.strokeStyle = "#ddd";
    ctx.lineWidth   = 25;
    ctx.stroke();

    var newEnd = (2.4-0.6)*(value/max) + 0.6;

    ctx.beginPath();
    ctx.arc(centerX,centerY,radius,0.6*Math.PI,newEnd*Math.PI)
    ctx.strokeStyle = "#aaa";
    ctx.lineWidth   = 25;
    ctx.stroke();


  }
```

首先，函数要用clearRect函数清空图层。需要注意的是，canvas知道自己的内部大小，因此对矩形的大小不需要进行硬编码。接下来，绘制背景圆弧作为仪表盘的背景。这里使用arc函数绘制部分圆，并且不是先绘制外层圆弧和内层圆弧，再填充结果，而是将lineWidth设置为较高的值，然后调用stroke方法生成而不是填充轨迹，从而生成背景。同时，还要对半径稍作调整，以使轨迹边缘不会超出canvas。

然后要以略深的颜色再次重复上述过程，从而表示仪表盘的读数。对于各特定度量的第二遍绘制的轨迹，会基于当前值和最大观察值来计算新的终点位置。如果已知仪表盘读数存在一个有限的范围，那就可以将这个最大值设置为范围的终点值。

然后将上述函数附加到数据事件，从而每当度量值有所改变时，仪表盘背景就可以更新：

```
var max = 100;
 $(document).on('data:'+name,function(event,data) {
   if(data > max) max = data;
   drawGauge(ctx,data,max,dims);
 });
```

向最初的示例仪表板应用加入上述代码，最终结果如图7-5所示。现在，每个元素后面都有一个仪表盘，仪表盘的读数会随着度量值的改变而更新。

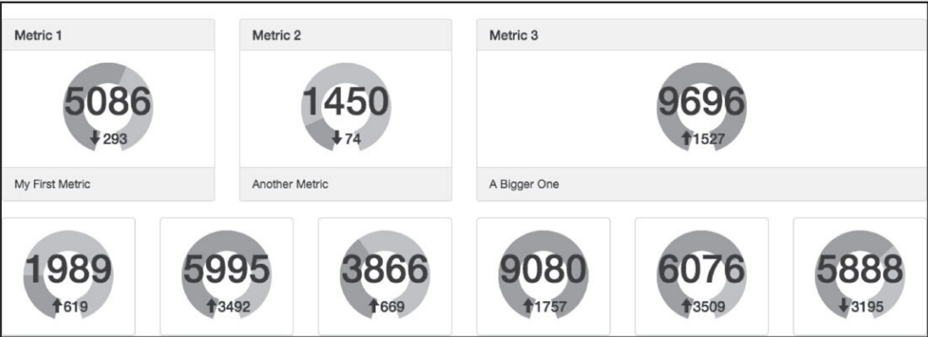


图　7-5

还有一个使用Canvas的例子，是在度量值背景中绘制更传统的图形。要将这两种可视化混合在同一个仪表板上，可以用第二个mixin类来处理图形，以dashboard/views/canvas_ex2.jade中的mixin为例：

```
mixin metricChart(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        canvas(data-canvas-chart="#{id}")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
          span.glyphicon(data-direction="#{id}")
          span(data-change="#{id}")= ""
  if footer
    div.panel-footer= footer
```

使用这个新的mixin，可以将第一行度量转换为“图形”模式，而不是“仪表盘”模式，这段代码位于dashboard/views/canvas_ex2.jade：

```
ock content
div(class="container")
  div.page-header: h1 Canvas Example 2
  div(class="row")
    div(class="col-lg-3")
      +metricChart("first","Metric 1","My First Metric")
    div(class="col-lg-3")
      +metricChart("second","Metric 2","Another Metric")
    div(class="col-lg-6")
      +metricChart("third","Metric 3","A Bigger One")
```

将一个不同的数据元素附加到图形canvas，就可以选择这些数据元素，并将其与其他绘制方法相关联。这里，渲染过程会生成一个数组，这个数组保存各个计数器发送的最后的maxValues数据元素，从而创建一个简单的区域图，该实现代码位于dashboard/public/javascripts/canvas.js：

```
var maxValues = 20;
function drawChart(ctx,values,max) {


  //Clear the gauage
  ctx.clearRect(0,0,ctx.canvas.width,ctx.canvas.height);

  var valHeight = ctx.canvas.height/max;
  var valWidth  = ctx.canvas.width/(maxValues-1);

  if(values.length < 2) return;
  ctx.beginPath();
  ctx.moveTo(0,ctx.canvas.height);
  for(var i=0;i<values.length;i++) {
    ctx.lineTo(i*valWidth,ctx.canvas.height - values[i]*valHeight);
  }
  ctx.lineTo(valWidth*(values.length-1),ctx.canvas.height);
  ctx.fillStyle = "#ccc";
  ctx.fill();
}
```

最后，将各元素放入数据数组中之后，就执行该绘制方法，从而生成图7-6。

```
$('*[data-canvas-chart]').each(function(i,elt) {
  fixup(elt);
  var name = $(elt).attr("data-canvas-chart");
  var ctx  = elt.getContext("2d");

  var chart = [];
  var max   = 100;
  $(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;

    chart.push(data);
    if(chart.length > maxValues) chart.shift();

    drawChart(ctx,chart,max);
  });

});
```

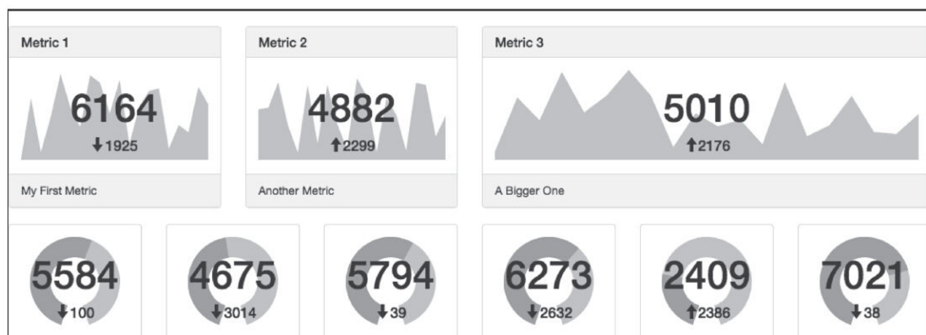


图 7-6

尽管比较粗糙，但这个例子说明，使用canvas元素渲染简单图形是很容易的。这里只是用canvas元素小试牛刀。还有许多其他的绘制工具，可以用来使元素生成的可视化更有吸引力。

7.2.1.2 使用内联SVG

与canvas元素不同，可伸缩矢量图形（Scalable Vector Graphics，SVG）使用非常类似于HTML的DOM来定义要渲染的图像。SVG是一个比较古老的标准，可以追溯到1999年左右。几乎所有的浏览器都支持SVG，Internet Explorer的支持要滞后一些（微软曾经向W3C提交过一个名为矢量标记语言（Vector Markup Language，VML）的标准与之竞争），但自IE 9之后，也开始在一定程度上支持SVG。

与依赖分辨率的canvas不同，SVG是与分辨率无关的标准，它能在任何分辨率下渲染。SVG在移动设备上特别有用，因为移动设备的用户常常要对界面的某些部分使用放大和缩小操作，以放大触摸目标。

由于内联SVG是DOM的一部分，因此使用起来要比canvas更方便，因为可以用层叠样式表（Cascading Style Sheet，CSS），像对其他元素那样对它套用样式。这样界面的样式就容易保持一致，并且与设计者进行协作也更容易。

这个文档模型自身的特点折射了其历史渊源，它深受PostScript绘图模型影响，并在一定程度上受到了微软公司提出的VML的影响。它的原始对象很少：<circle>、<rect>、<polygon>和<path>元素与分组（<g>）元素等组织型元素一起，就构成了一幅图像。SVG也有自己的<text>元素，但它们通常不具备像HTML那么强的布局控制能力。

使用这些元素，就可以在mixin中静态地构建上一节的仪表盘可视化，该代码位于dashboard/views的jade文件中：

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        svg(data-counter-gauge="#{id}")
          path(class="back")
          path(class="front")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
          span.glyphicon(data-direction="#{id}")
          span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

然后在CSS中设计<svg>元素中的轨迹元素的样式，以生成所需要的效果：

```
path.back {
  fill: none;
  stroke: #ccc;
  stroke-width: 25px;
}
path.front {
  fill: none;
  stroke: #ccc;
  stroke-width: 25px;
}
```

如果要绘制圆弧，就会相应地修改每个轨迹元素的d属性。与用canvas时不同，只需要以最大长度绘制back元素一次。每当值更新时，front元素会被public/javascripts/svg.js中的下列函数所修改：

```
$('*[data-counter-gauge]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-gauge");

  drawArc($(elt).children(".back"),dim,1);
  var front = $(elt).children(".front");
  var max   = 0;
  $(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;
    drawArc(front,dim,data/max);
  });
});
```

主要工作是对轨迹元素中的圆弧的计算。这是在drawArc函数实现的，其中用到了三角函数。这段实现代码位于public/javascripts/svg.js，：

```
var mid = (2.4-0.6)*0.56 + 0.6;
function drawArc(elt,dim,pct) {
  var cx = dim.width/2;
  var cy = dim.height/2;
  var r  = Math.min(cx,cy) - 12.5;

  var angle  = ((2.4-0.6)*pct + 0.6)*Math.PI;

  var sx = cx + r*Math.cos(0.6*Math.PI);
  var sy = cy + r*Math.sin(0.6*Math.PI);

  var ex = cx + r*Math.cos(angle);
  var ey = cy + r*Math.sin(angle);
  elt.attr("d",["M",sx,sy,"A",r,r,0,
    angle <= mid*Math.PI ? 0 : 1,1,
    ex,ey].join(" "));
}
```

与许多场景描述语言类似，便于使用并不是SVG的设计目标。因此，尽管SVG中绘制圆弧的命令很强大，甚至远超多数用户的需要，但它的函数使用起来却像上面的例子一样繁琐，因此在文档中直接使用SVG并不方便。不过，你可以使用JavaScript库，来简化使用过程，例如用下一节介绍的D3.js。

7.2.2 数据驱动文档：D3.js

如果纯手工编写可视化，你很快就会感到枯燥乏味，当要进行复杂渲染时更是如此。正如我们在上一节所看到的，即使是表面上很简单的结构，都可能必须进行许多计算，并且有的选择也很难记。

数据驱动文档JavaScript库（Data-Driven Documents JavaScript library，简称D3.js或D3）致力于提供丰富的可视化操作原语集，用来简化渲染中的大部分计算。此外，它还提供了一种机制，以方便地将数据绑定到可视化，从而使我们可以快速构建出前面几节的示例。

本节介绍D3.js库的基础知识及用法。其中包括基本的操作和渲染、布局，以及高层构件，如坐标轴和地图。它的工具包中还包含了可视化的交互功能，不过这部分内容不在本书的范围之内。

7.2.2.1　选择、插入和删除元素

与jQuery及其他JavaScript工具包类似，D3的核心特性是选择。实际上，D3使用的选择器语言与jQuery非常相似。例如，要选择文档中的所有＜div＞元素，应使用d3.selectAll（“div”）。要选择单个元素，例如＜body＞，应使用d3.select（“body”）。另外，与jQuery类似，d3.selectAll（“div.blue”）选择的是类属性包含blue的所有＜div＞元素，而d3.select（“div#blue”）选择的是id属性为blue的＜div＞元素。还可以将这些操作串接起来，这样，先调用select，再接着调用selectAll，就可以查找第一个选择操作结果的所有子元素。

选择某个元素之后，最常见的操作是，向该元素加入另一个对象，作为其子元素。例如，下列函数用来对仪表板进行初始化。在固定了包装器类的位置之后，可以用D3创建一个＜svg＞元素，这个元素可以用于进一步渲染。该函数位于dashboard/public/javascripts/d3.js：

```
$('*[data-counter-gauge]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-gauge");
  var svg = d3.select(elt).append("svg");
```

也可以用insert方法来插入元素，在该方法的输入参数中，除了要插入的元素，还有一个before语句。例如，要将一个元素插入到子元素列表的开头，而不是末尾，应使用insert（“＜div＞”，“：first-child”）。

最后，可以使用remove（）方法，将所选的元素从文档中删除。

7.2.2.2　属性和样式

选择某个元素之后，可以用选择结果类的attr和style方法对其修改。在简单情况下，这些方法与对应的jQuery方法很相似，会修改元素的对应内容。在仪表盘的例子中，使用它们对前后的仪表盘对象建立相应的类：

```
var g = svg.append("g")
  .attr("transform",
    "translate("+(dim.width/2)+","+(dim.height/2)+")");
var back  = g.append("path").attr("class","d3back");
var front = g.append("path").attr("class","d3front");
```

这里，将轨迹元素加入<g>元素中，而不是直接加入<svg>元素。这个分组元素支持对所有组进行变换操作，例如平移。这里使用的平移将轨迹放到中心位置，以使图形生成器可以应用到每个轨迹。

7.2.2.3　图形生成器

在D3中，上一节那样的轨迹元素通常是与图形生成器结合在一起的，而不是人工描述的。这些生成器用来让d属性的设置变得更容易。D3内置有各种轨迹生成器：

·d3.svg.path.line生成简单的线形轨迹。

·d3.svg.path.area生成与第7.2.1.1节中的绘图示例类似的区域，而不是线。

·d3.svg.line.radial生成极坐标线形轨迹。

·d3.svg.arc生成圆弧轨迹，常用于饼图和环形图。

·d3.svg.symbol用来生成特定位置的符号，例如散点图中的符号。

·d3.svg.chord用来生成以二次贝塞尔曲线连接两个圆弧构成的闭合图形。

所有的图形生成器都会返回一个函数，这个函数可以用特定生成器的参数来定制。这些参数可以设置为常数，也可以设置为函数，从而可以根据输入生成相应输出。

例如，要创建上一节的仪表盘，可以用d3.svg.arc生成器来简化该过程。该生成器有四个不同的参数，可以根据数据对其进行设置：内半径和外半径、起点角和终点角。这里，前三个参数都是常数：

```
var r     = 0.5*Math.min(dim.width,dim.height) - 12.5
var arc   = d3.svg.arc()
  .innerRadius(r-12.5).outerRadius(r+12.5)
  .startAngle(1.1*Math.PI);
```

最后一个参数终点角取决于传输的数据，因此为它赋予一个函数，而不是常数：

```
var max = 1;
arc.endAngle(function(d,i) {
  if(max < d) max = d;
    return 1.1*Math.PI + 1.8*Math.PI*(d/max);
});
```

将该参数的最大值置为1，一次性地绘制背景元素，然后每当有新的值就重新绘制前景元素：

```
back.attr("d",arc(1));
$(document).on('data:'+name,function(event,data) {
  front.attr("d",arc(data));
});
```

除了易于实现，由圆弧生成器（以及相应的其他生成器）生成的轨迹是区域，而不是线。这意味着这些轨迹可以有单独的填充和轮廓样式，而在以前的例子中，为了简化渲染，仅使用了轮廓宽度作为参数。

7.2.2.4 将选择操作与数据相连接

当数据比较简单时，可以直接使用D3，这是开发快速粗糙可视化的简单方法。当数据变得复杂之后，D3就更能显示出它的威力，因为它的连接操作可以与选择操作符结合起来使用。

要将数据数组连接到选择操作，需要使用选择操作的data方法。这样得到的连接由三个子选择组成：

·更新（update）选择：文档中已经对应了数据数组中元素的元素组成的集合。

·进入（enter）选择：数据数组中（尚）没有对应文档元素的元素组成的集合。

·退出（exit）选择：在数据数组中没有对应元素的文档元素集合。

更新选择是data方法返回的默认选择。其他两个选择分别由enter（）和exit（）方法来得到。

常见的顺序是首先进行“进入选择”。该选择在两个方面比较特殊。第

一，它只影响append这样的文档操作方法。第二，当使用append（）或insert（）方法时，结果元素会被立即加入到“更新选择”中。这样，通过允许在“更新选择”中修改全部属性和样式，就可以减少重复代码。如果因为某种原因，新创建元素的样式设计是不同的（可能是为了强调），只需要在使用“进入选择”前修改“更新选择”。

“退出选择”通常是最无趣的选择，因为该选择唯一的操作往往就是从文档中删除元素。有时这会结合一定的动画（在7.2.2.7节讨论），从而使效果没那么突兀。

要查看这些选择的实际效果，实现一个条形图是一个好方法。这里，对mixin进行了少许修改，以应用data-counter-bar属性。由于在本章中我们曾经多次这样做过，因此在这个例子中我们略去了该代码。与仪表盘元素类似，每个条形图元素都附加有一个<svg>元素。为了方便设计条形图的样式，我们编写了一个类：

```
$('*[data-counter-bar]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-bar");
  var svg   = d3.select(elt).append("svg")
  .attr({
    class:"barchart",
    width:dim.width,
    height:dim.height
  });
```

与折线图的例子类似，当新元素到达时，会维护一个元素数组。该数组会被绑定到选择，并渲染到条形图中：

```
var dataLen= 30, max = 0;
var values = [];
$(document).on('data:'+name,function(event,data) {
  if(data > max) max = data;
  values.push(data);
  if(values.length >= dataLen) values.shift();
```

然后将该数组与包含全部矩形的选择相连接。并将一个新矩形加入到选择（如果需要的话）中，这样所有矩形都会更新为合适的位置和高度。加入矩形时要设置宽度，这是因为它不会在可视化过程中发生变化。最终生成的仪表板如图7-7所示，实现代码位于dashboard/public/javascripts/d3.js和dashboard/views/d3_ex1.jade：

```
    var update = svg.selectAll("rect").data(values);
    //Add a new rectangle for each piece of data
    update.enter().append("rect").attr("width",dim.width/(dataLen-1))
    update
    .attr("x", function(d,i) { return i*dim.width/(dataLen-1); })
    .attr("y", function(d) { return dim.height - dim.height*(d/max); })
    .attr("height", function(d) { return dim.height*(d/max); })
  });
```

图 7-7

要重新创建像原来的canvas例子那样的折线图，需要另外的方法。这时，应该将单个元素（轨迹）绑定到整个数据数组。唯一使用多个轨迹元

素的情况是有多个序列要渲染的时候。此时，可以用datum方法将整个数组绑定成一个元素。

7.2.2.5　刻度和坐标轴

在前面的例子中，大量代码被用来将输入域变换为输入范围。为了简化这个过程，D3有许多内置的刻度函数：

·d3.scale.linear执行从输入域到输出范围的线性变换。这是最常用的刻度。

·d3.scale.sqrt、d3.scale.pow和d3.scale.log执行从输入域到输出范围的幂函数变换。

·d3.scale.quantize将连续的输入域转换为离散的输出范围。

·d3.scale.identity为简单的线性等值刻度。

·d3.scale.ordinal将离散输入域转换为连续输出范围。

·d3.scale.category10、d3.scale.category20、d3.scale.category.20b和d3.scale.category.20c构造一个有序刻度，分别转换为10到20个色彩类别。

多数情况下，这些刻度用来简化范围的计算。例如，可以使用x和y坐标的线性刻度，对上一节中的条形图例子做进一步简化。x的刻度是静态的，输入域处于0和数组中元素数量（这里是30）之间。y的刻度可能随着数据的加入发生变化，因此每一步都要进行检查，来确定是否需要对域进

行更新：

```
var y = d3.scale.linear().domain([0,1])
  .range([0,dim.height]);
var x = d3.scale.linear().domain([0,dataLen-1])
  .range([0,dim.width]);

$(document).on('data:'+name,function(event,data) {
  if(data > y.domain()[1]) y.domain([0,data]);
```

然后就可以通过调用x和y刻度，从而对更新步骤中的计算进行更新：

```
update

  .attr("x", function(d,i) { return x(i); })
  .attr("y", function(d) { return dim.height - y(d); })
  .attr("height", function(d) { return y(d); });
```

刻度的另外一个用处是，简化向D3.js的可视化加入坐标轴的过程。与多数其他D3元素类似，坐标轴是用一个生成器来管理的，该生成器操作<svg>元素或<g>元素。坐标轴函数向文档中加入大量新元素，包括表示刻度标记和刻度值的线。使用坐标轴最简单的方法是使用分组元素，因为这样在图形中移动起来就会容易一些。开始时，需要对刻度做细微调整，从而为坐标轴的渲染腾出空间：

```
var y = d3.scale.linear()
  .domain([0,1])
  .range([dim.height-20,0]);
var x = d3.scale.linear()
  .domain([0,dataLen-1])
  .range([60,dim.width-10]);
```

注意，y轴的刻度保留了其输出范围。这是因为（0，0）坐标实际上位于左下角，而不是左上角。这也意味着在绘制条形图时必须保留“高度”和“y”属性。由于x轴是静态的，因此可以立即将其渲染到分组元素中，该分组元素则会被平移到可视化的底部：

```
svg.append("g")
.attr("class","x axis")
.attr("transform","translate(0,"+(dim.height-20)+")")
.call(d3.svg.axis()
  .orient("bottom")
  .scale(x)
);
```

y轴的定义也是类似的，但只要域发生变化，它就要被更新，因此不直接将其应用到分组元素：

```
var yg    = svg.append("g")
.attr("class","y axis")
.attr("transform","translate(60,0)");
var yaxis = d3.svg.axis()
  .ticks(5).orient("left")
  .scale(y);
```

只要域被更新，就重新绘制y轴，以反映所发生的变化：

```
if(data > y.domain()[1]) {
  y.domain([0,data]);
  yg.call(yaxis);
}
```

在使用一点CSS样式后，该条形图就应该如图7-8所示。该实现代码位于dashboard/views/d3_ex2.jade：

```
.axis text { font: 10px sans-serif; }
.axis path, .axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
```

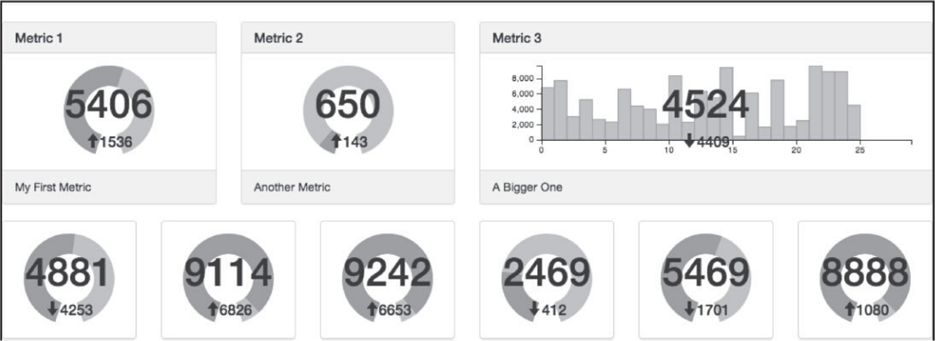


图　7-8

7.2.2.6　布局

许多情况下，当要对数据进行可视化时，需要进行一定程度的操作使其适于渲染。即使像饼图这样简单的图形，都需要计算每个分类的起点角和终点角。

为了应对这些更复杂的可视化，D3提供了一系列布局生成器。布局生成器通常应用于数据，然后再用来构建可视化，而不是像轨迹生成器和坐标轴生成器那样直接生成文档元素。

有许多实现各种布局的D3插件，但也有一些内置的插件。许多插件实现的是层次结构或图形结构的布局。当数据是静态的时候，这类布局挺有用，但对于实时数据的可视化用处不大，因此这一节我们不会讨论这类布局。

实时数据可视化最频繁使用的布局是饼状布局和堆叠布局。这两种布局分别用来构建饼图或环状图以及堆叠面积图。例如，要构建图7-9这样

的图，首先要构建一些示例数据：

```
var values = d3.range(10).map(function() {
    return bumpLayer(100);
});
```

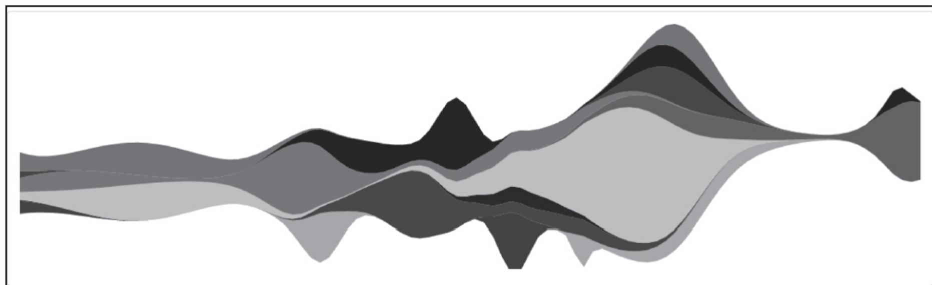


图 7-9

在D3文档的很多地方都用到了bumpLayer函数，用来生成测试数据。在这里使用该函数，是为了给这个例子快速提供数据。接下来，使用stack布局来堆叠数据，并计算最大值，这样我们可以获得刻度：

```
var stack = d3.layout.stack()
  .offset("wiggle");

values = stack(values);

var max = 0;
for(i in values) {
  var m = d3.max(values[i],function(d) { return d.y + d.y0; });
  if(m > max) max = m;
}
```

然后构建x轴、y轴的刻度以及颜色范围。我们使用area轨迹生成器来渲染stack布局的输出：

```
var x     = d3.scale.linear().domain([0,99]).range([0,width]);
var y     = d3.scale.linear().domain([0,max]).range([height,0]);
var color = d3.scale.linear().range(["#000","#ccc"]);

var area  = d3.svg.area()
.x(function(d) { return x(d.x); })
.y0(function(d) { return y(d.y0); })
.y1(function(d) { return y(d.y0 + d.y); });
```

最后，将图形生成器和布局数据应用到轨迹选择中，从而生成最终的输出。实现代码参见dashboard/views/d3_ex3.jade：

```
svg.selectAll("path").data(values)
  .enter().append("path")
  .attr("d",area)
  .style("fill",function(d) { return color(Math.random()); });
```

7.2.2.7 动画

对于D3，要介绍的最后一个内容是动画功能。是否容易加入动画效果往往是区分普通界面和优秀界面的因素之一。

通常来说，根据具体情况的不同，不论目的是要吸引还是转移用户的注意力，动画都应该谨慎使用。例如，如果对仪表板上度量数值的变化加上动画效果，那就会将用户的注意力吸引到数值的变化上来，因为如果不加动画效果，人眼不容易察觉这种变化。然而，如果允许仪表盘自身的显示可以“跳动”，这反倒会让人分心。为了应对这种分心效应，并使用户关注新的值，通常会转而使用简单的“渐变”动画。这种动画使仪表盘的显示随着时间的移动变得平滑，这样就不那么容易让用户分心。

对于简单的动画，基本上不需要做什么。例如，如果要为前面例子中的条形图增加动画效果，只需要加入一个transition函数，以进行元素间的平滑插值：

```
update
  .transition()
    .attr("x", function(d,i) { return x(i); })
    .attr("y", function(d) { return y(d); })
    .attr("height", function(d) { return (dim.height-20) - y(d); });
```

transition方法有很多特性，例如easing方法或duration，但对于许多可视化来说，平滑插值就足够了。

只有在极个别的情况下，才需要更复杂的过渡效果。仪表盘的例子恰好属于这种情况。如果需要在仪表盘中直接应用过渡效果，那就要在页面的笛卡尔坐标系下进行插值，而不是仪表盘的极坐标系下插值。如果要修改仪表盘使其支持动画，首先必须对轨迹进行修改，才能使用数据元素：

```
var back  = g.append("path").datum({endAngle:1})
 .attr("class","d3back")
 .attr("d",arc);
 var front = g.append("path").datum({endAngle:0})
 .attr("class","d3front")
 .attr("d",arc);
```

之所以这样做，是因为插值过程需要能修改endAngle变量。具体的插值过程是通过attrTween反复应用插值来实现的。具体过程实现于dashboard/public/javascripts/d3.js中的arcTween函数：

```
function arcTween(transition,newAngle) {
  transition.attrTween("d",function(d) {
    var interpolate = d3.interpolate(d.endAngle,newAngle);
    return function(t) {
      d.endAngle = interpolate(t);
      return arc(d);
    }
  });
}
```

该函数反复计算插值的角度，然后以该插值角度调用圆弧生成器，从而生成新函数。由于没有调用datum方法，就没有定义d变量，因此没有

地方存储下一次插值的结果。这样，更新步骤只需要在新角度到达时用该角度调用arcTween函数：

```
$(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;
    front.transition().call(arcTween,data);
 });
```

7.2.3 高层工具

尽管D3的功能非常强大，但它的粒度过细。虽然已经有一些抽象接口，但建立基本的可视化仍然比较耗时。为此，有的人建立了一些高层软件包，这些包建立在D3之上，支持简单可视化的快速开发。本节介绍其中两种有趣的软件包。这两个包都不是广泛使用的，但能使你在建立仪表板可视化时站到一个较高的起点。第一个包NVD3很容易找到，对于新手来说也更容易上手。第二个包Vega代表了一类有趣的方法，这类方法将渲染、业务逻辑和数据分离开来。

7.2.3.1 NVD3

NVD3已经使用了很多年，但仍然处于活跃的开发状态。撰写本书时，它的beta版本号为1.1.10。它是“dashing”仪表板库的后端库。“dashing”库是来自Shopify公司的简单的流式仪表板框架，它使用Sinatra框架（而不是Node），用Ruby语言编写。

图7-10展示了与上一节类似的条形图，该图是用NVD3而不是直接用D3来绘制的。图中还展示了一个类似的可视化，这个可视化使用的是折线图，而不是条形图。建立条形图时，首先要定义NVD3图形，然后将其附

加到SVG元素。代码位于dashboard/public/javascripts/nv.js：

```
$('*[data-counter-bar4]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-bar4");

  var chart = nv.models.historicalBarChart()
  .margin({left:50,bottom:20,right:20,top:20})
  .x(function(d,i) { return i; })
  .y(function(d,i) { return d; })
  .transitionDuration(250);

  chart.showXAxis(true);
  chart.yAxis
  .axisLabel("Value");


  var values = [];
  var dataLen   = 30;
  d3.select(elt)

 .append("svg")
 .datum([{values:values,key:"Data",color:"#ccc"}])
 .transition().duration(0)
 .call(chart);
```

代码中使用了datatum命令，这是因为NVD3使用的是序列，而不是数据，这与图形生成器很相似。如上述代码所示，由于使用序列，还可以定义序列标题和特定序列的颜色。序列的值刚开始是空的，但由于JavaScript是基于引用的语言，因此不需要重新设置数据元素，就可以在随后更新数组。定义图形之后，在数据数组被修改时更新图形就很容易了，调用update方法即可：

```
$(document).on('data:'+name,function(event,data) {
  values.push(data);
  if(values.length >= dataLen) values.shift();
    chart.update();
});
```

NVD3接口非常标准，这使得切换图形类型易如反掌。要将图7-10中

的条形图改为线性图，只需要修改一行代码：

```
var chart = nv.models.lineChart()
```

D3 Example 4

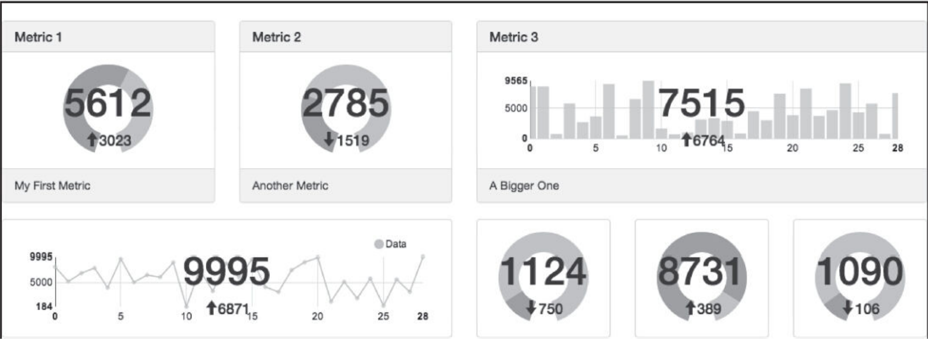


图 7-10

折线图的数据序列有一个可选的area参数，可以用来将折线图变成区域图。参数标准化使得对不同可视化进行试验变得非常简单。另外，通过修改datum语句，在同一个图中渲染多个序列也非常简单。

很显然，NVD3可以处理流数据，但它主要还是用于静态交互式可视化。因此，它包含了许多额外的交互特性，例如提示框、缩放和聚焦等交互式可视化环境中的常见特性。但鉴于多数实时环境不是交互式的，本章不讨论这些特性。

7.2.3.2 Vega.js

开发Vega软件包的初衷是使可视化的描述易于修改。Vega.js不是像VND3那样用代码来描述可视化，而是选择用可视化的JSON表示来描述。Vega开发人员将该格式称为“声明式可视化语法”。

Vega可视化首先要定义图形规格，这个规格负责给出图形的抽象可视化。规格由若干个部分组成。第一部分通常是图形渲染的一些信息。由于图形经常会被重用，因此这些信息即使有，也通常比较少。对所有图形来说，最常见的是页边距信息：

```
var spec = {
  padding: {top: 10, left: 30, bottom: 20, right: 10},
```

接下来是规格的数据定义部分。这一部分定义可视化中会用到的各种数据序列。由于数据来自实时系统，会在随后加入进来，因此只需要将序列的数据部分留为空即可：

```
data:[
  {name:"values"}
 ],
```

最后，对可视化进行描述。与低层次的D3图形非常类似，规格使用与内部使用的D3生成器相似的参数，用来定义刻度以及坐标轴：

```
scales:[
{name:"x",range:"width",
    domain:[0,29] },
{name:"y",range:"height",nice:true,
    domain:{data:"values",field:"data"}},
],

axes:[
{type:"x",scale:"x"},
{type:"y",scale:"y"}
],
```

上述代码中，要注意的主要方面就是数据序列与domain元素中的刻度的链接。这个链接结构称为DataRef，通常由一个数据序列和一个字段访问器构成。前者由data元素所标识，后者由field元素定义。当数据被加入到Vega中时，假设数组如下所示：

```
[0,1]
```

该数组被转换为“与Vega兼容”的格式：

```
[{index:0,data:0},{index:1,data:1}]
```

字段访问器对这个转换后的数据进行操作，这时需要保证数据包含index和data字段才能正确运行。可以使用常见的JavaScript点表示法，来访问更复杂的数据结构。

在规格中定义“标记”（marks），可以为给定序列规定所绘制图形的类型。与NVD3类似，如果定义多个标记，就可以在一个图形中绘制多个元素。这里，只渲染了一个rect标记，以绘制条形图：

```
marks:[{
  type:"rect",
  from:{data:"values"},
  properties:{
    enter:{
      fill:{value:"#ccc"},
      stroke:{value:"#aaa"}
    },
    update:{
        x:{scale:"x",field:"index"},
        y:{scale:"y",field:"data"},
        y2:{scale:"y",value:0},
        width:{scale:"x",value:1}
      }


    }
  }]
};
```

同样要注意用来绑定参数和数据的DataRef。还要注意scale元素，它被用来将参数关联到刻度，除此之外，标记的定义看起来就似曾相识了，因为从本质上说，它们就是前面使用的D3命令的JSON表示。

要使用该规格，必须首先将其解析为chart对象。该对象是在回调中返回的，因为可以向解析器传递URL，而不是直接传递JSON对象。这使得我们可将规格与代码分开存储。得到chart对象之后，可以运行该对象来生成view对象，该对象用来渲染场景。Vega有两个默认渲染器，一个基于canvas，另一个基于SVG。这里使用的是后者，但默认的是canvas渲染器：

```
vg.parse.spec(spec,function(chart) {

  $('*[data-counter-bar4]').each(function(i,elt) {
    var dim = fixup(elt)
    var name = $(elt).attr("data-counter-bar4");

    var view = chart({el:elt})
    .renderer("svg")
    .width(dim.width-60)
    .height(dim.height-30)
    ;
    var values = [];
    var dataLen    = 30;
    var x       = 0;
    $(document).on('data:'+name,function(event,data) {
      values.push(data);
      if(values.length >= dataLen) values.shift();
      view.data({values:values}).update();
    });
  });


});
```

在上述代码中，还使用了Vega的动态数据特性。View对象的该方法的输入参数是一个带有键的对象，该键对应了规格中定义的表。这些键的值覆盖了视图对象的表定义中的对应键值。最终结果如图7-11所示，该实现代码位于dashboard/public/javascripts/vega.js。

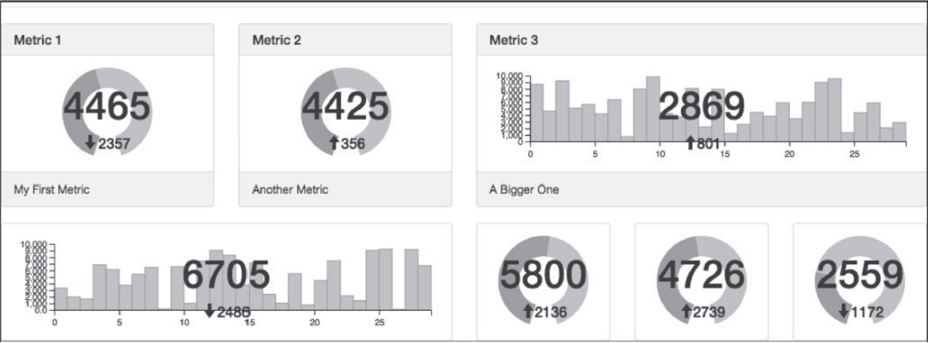

D3 Example 5
Metric 1
4465
2357
My First Metric
Metric 2
4425
356
Another Metric
Metric 3
2869
801
A Bigger One
6705
2486
5800
2136
4726
2739
2559
1172

图　7-11

7.3 移动流应用

当今世界，越来越多的应用趋向移动化，流数据应用也不例外，尽管这类应用在较大的平板电脑上要比较小的手机上更有用。

本章开发的基础架构可以适用于移动环境。对于数据传输来说，消耗电池电量最严重的就是那些频繁使用Wi-Fi或手机无线功能的应用。轮询方法会频繁使用无线功能。从经验上来说，SSE和WebSocket似乎对电池续航时间影响较小。此外，这两种协议都支持自动重连，因此在不够稳定的移动通信环境中，使用这两种协议是不错的选择。

在渲染的前端，用来开发仪表板的Bootstrap 3框架是“移动优先”框架，它具有内置的“响应”特性。这意味着在手机和平板电脑上，仪表板会重新调整，以使移动平台上的可用空间最大化。图7-12展示了仪表板的响应式重排示例。

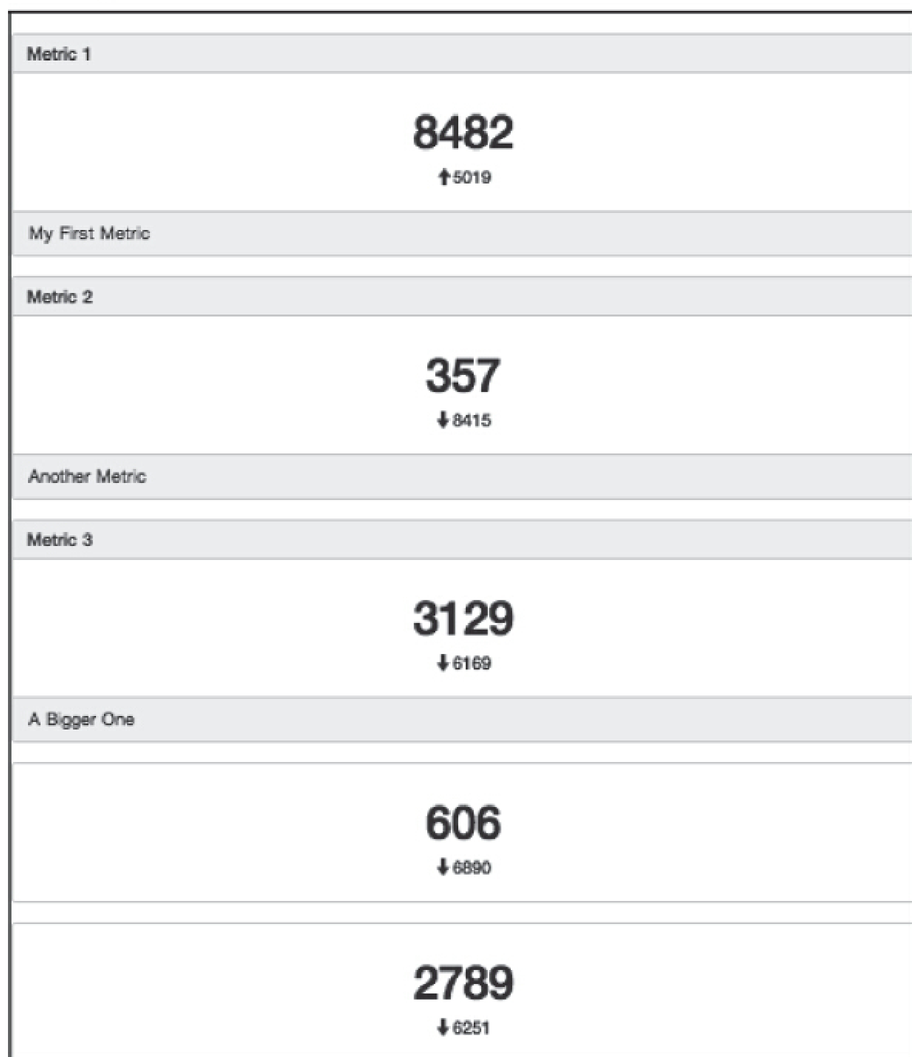


图 7-12

对可视化渲染来说，所有主要的移动浏览器都支持Canvas和内联SVG。即使是移动版的Internet Explorer和黑莓浏览器也支持SVG。由于SVG基于向量进行渲染，它也非常适合处理高分辨率渲染，这样的高分辨

率（以及较好的缩放效果）在现代平板电脑和手机中很常见。SVG（与Bootstrap不同）唯一的问题是，它本身不是“响应”式的。图7-13演示了当SVG未能在大小变化时及时调整的样子。

图像仍然是以原来的视角绘制的，但现在图形与面板的尺寸不相匹配。为了解决这个问题，必须捕获大小调整事件（resize event），并用其更新依赖元素大小的元素。例如，为了在发生大小调整事件时更新仪表盘元素，需要修改每个元素：

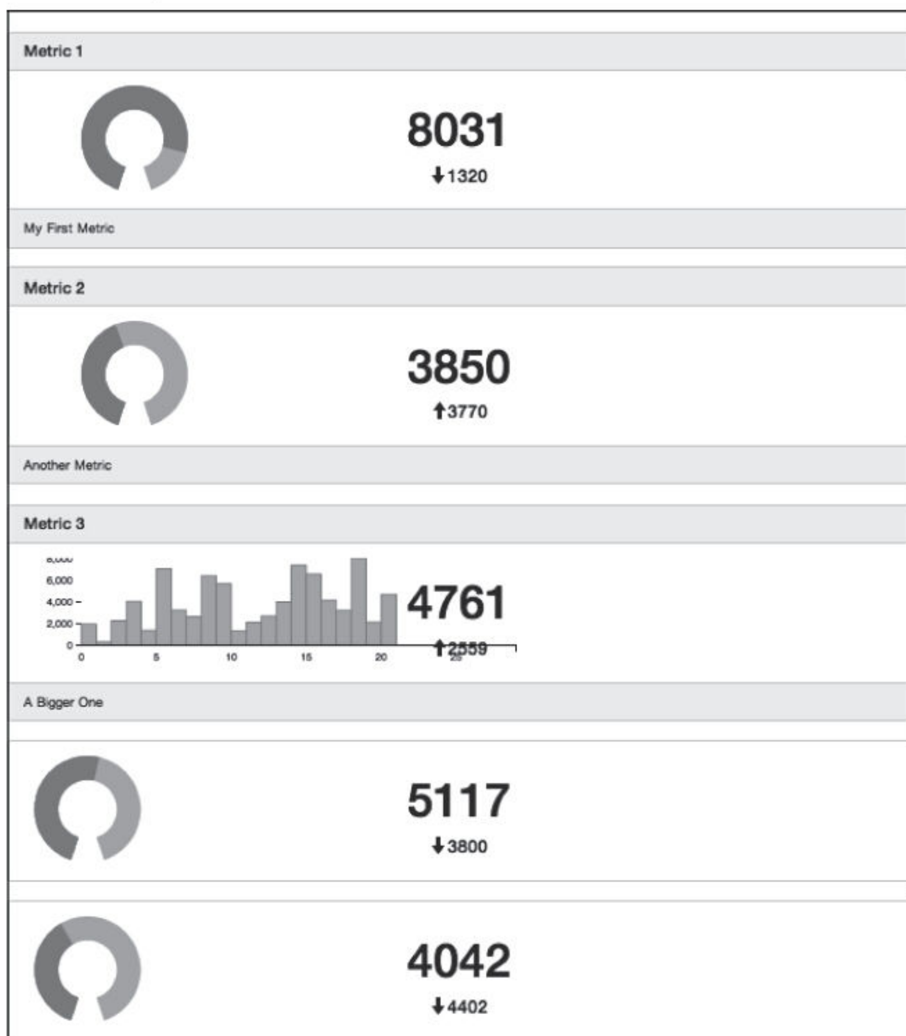

D3 Example 2
Metric 1
8031
1320
My First Metric
Metric 2
3850
3770
Another Metric
Metric 3
6,000
4,000
2,000
0
0
5
10
15
20
4761
2559
A Bigger One
5117
3800
4042
4402

图 7-13

```
$(window).resize(function() {
  dim = fixup(elt);
  g.attr("transform",
    "translate("+(dim.width/2)+","+(dim.height/2)+")");
    r = 0.5*Math.min(dim.width,dim.height) - 12.5;
    arc.innerRadius(r-12.5).outerRadius(r+12.5);
    back.transition().duration(0).call(arcTween,max);
  });
```

这里，要重新调整canvas的大小，并从fixup函数得到新尺寸。接下来，将圆弧平移到新位置。由于该元素可能已经改变了最小尺寸，因此必须重新计算内外半径。最后，在新的位置重新绘制后面的圆弧。还可能还要重绘前面的圆弧，但由于它往往是由流数据来更新的，因此在这里我们不重绘前圆弧。

这实际上要比用Canvas简单一些。因为如果是canvas渲染，每一遍都要完全重新绘制。可以直接为大小调整事件设置一个标志，告诉canvas下次重绘时更新大小。

7.4 总结

针对实时数据交付，本章介绍了建立基于Web的应用的基础知识。在下一章中，我们基于这些概念，向数据和可视化中加入时间元素。

除此之外，本书剩下的部分实际上并没有涉及交付和可视化机制。并不是说对于这个主题没有其他的内容可讲。拿可视化组件来说，本章很少触及使用这些框架能做什么。俗话说得好，“唯一的限制就是想象力”。实际上，在可视化这个领域里，有大量激动人心的项目。第一个例子就是Mike Bostock的网站（http://bost.ock.org/mike/ ），里面有大量深入介绍D3的教程。如果你正在学习怎样对地理数据可视化，那么你会发现这个网站特别有用，因为显然他本人特别爱好地图学，同时D3对地理数据可视化的支持也令人印象深刻。Flowing Data网站（http://flowingdata.com ）中也包含了大量可视化的资源。并不是里面所有的资源都适合于实时流数据，但其中的技术当然是适用的。

第8章　精确的聚集计算和交付

既然已经可以用基础架构来采集、处理、存储和交付流数据，现在是时候将这些功能放在一起解决具体问题了。对大多数应用来说，核心问题是对流中的数据进行聚集计算，这就是本章的主题。

让我们从最基本的聚集计算开始，即对各种数据元素的计数、求和。在主流的几个流处理框架中，对这个基本的任务提供了各种不同层次的本地支持。其中提供最高级支持的应该是Storm的Trident语言，不过前提是用户情愿使用Trident。Samza通过使用其KeyValueStore接口支持一定的内部聚集计算，但它将外部聚集和交付留给用户实现。基本的Storm拓扑当然没有什么与聚集计算相关的原语，因此要由用户来实现拓扑的所有细节。在本章中，我们使用了一个常见的例子来介绍聚集计算，这个例子涵盖了所有这三种框架。

多数实时分析都或多或少地与时间序列分析相关。本章也介绍了一种针对时间序列的多分辨率聚集计算方法，以供后面的时间序列交付使用。此外，由于数据传输机制的原因，事件可能会发生乱序，因此，这种多分辨率方法不是简单地以固定间隔生成结果，而要依靠记录的时间戳来保证计算结果的准确性。

最后，在将数据采集和聚集到事件序列中后，就必须将其交付给客户端。使用上一章介绍的技术就可以将时间序列数据交付给客户端来渲染。对于时间序列数据，有多种不同的渲染选择，其中最流行的是简单的带状图（strip chart），因为它易于理解。另一个选择是地平线图（horizon

chart），这种图可以同时显示许多相关联的时间序列。当数据交付速度很快时，使用标准的渲染技术有时可能会太慢。如果想要以非常高的更新速率来绘制图形，可以使用高性能的渲染技术，不过要牺牲图形绘制的灵活性。

维基百科文章流

维基百科通过IRC（Internet Relay Chat）通道向全世界有需要的人开放文章流。任何时候，维基百科上都有海量的文章，因此对于实时流应用的初学者来说，它是一个很好的“测试”数据源。在本章中我们会一直使用该数据流作为工作示例。

要获取该数据，需要一个应用来监视IRC数据流。正好第5章中的Samza项目包含了一个维基百科IRC reader，它会将流输入Kafka队列中。这个IRC reader是Hello Samza项目的一部分。要运行它，首先要从Github中检出入门级代码，并启动其中包含的Samza grid：

```
$ git clone https://github.com/apache/incubator-samza-hello-samza.git
$ cd incubator-samza-hello-samza/
$ ./bin/grid bootstrap
.. Output Removed ...
EXECUTING: start zookeeper
JMX enabled by default
Using config: /Users/bellis/Projects/incubator-samza-hello-
  samza/deploy/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
EXECUTING: start yarn
EXECUTING: start kafka
```

Samza是一个非常年轻的项目，它的代码库变化很快。因此，公共资源库的变化可能导致本书中的例子无法运行。为了应对这种情况，在本书的代码中，还包含了撰写本书时incubator-samza和hello-samza项目的副

本。要使用这些副本，只需要将归档文件解压，并将incubator-samza项目复制到samza下载目录。在多数的类Unix系统中，该目录是~/.samza/download。如果一切顺利，./bin/grid bootstrap命令就会像上面所讲的那样运行起来。

这样会在本地启动YARN控制台，从而可以对本章中的应用进行开发和测试。接下来，构建和部署Hello Samza项目的第一部分，开始对维基百科的文章进行聚集计算：

```
$ mvn clean package
... Omitted Output ...
 [INFO] Reactor Summary:
[INFO]
[INFO] Samza Parent ................................ SUCCESS [2.707s]
[INFO] Samza Wikipedia Example ..................... SUCCESS [17.387s]
[INFO] Samza Job Package ........................ SUCCESS [1:24.473s]
[INFO] ---------------------------------------------------------------
[INFO] BUILD SUCCESS
  $ mkdir -p deploy/samza
  $ cd deploy/samza
  $ tar xvfz \
  > ../../samza-job-package/target/samza-job-package-0.7.0-dist.tar.gz
  $ ./bin/run-job.sh \
  > --config-factory=\
  > org.apache.samza.config.factories.PropertiesConfigFactory \
  > --config-path=\
  > file://`pwd`/config/wikipedia-feed.properties
```

如果一切正常，在YARN控制台中，应该有一个状态为RUNNING的应用程序（默认情况下，控制台位于http://localhost:8080 ）。现在应该也可以使用Kafka控制台消费者通过监视wikipedia-raw主题来查看原始的文章流：

```
$ ../kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost --topic wikipedia-raw
```

稍候片刻，原始的文章流就会开始从Kafka流中输出出来：

```
{"raw":"[[Special:Log/newusers]] byemail  * Callanecc *  created new
  account User:DeaconAnthony: Requested account at [[WP:ACC]],
  request #109207","time":1382229037137,
  "source":"rc-pmtpa","channel":"#en.wikipedia"}

{"raw":"[[User talk:24.131.72.110]] !N
    http://en.wikipedia.org/w/index.php?oldid=577910710&rcid=610363937
  * Plantsurfer * (+913) General note: Introducing factual errors on
  [[Prokaryote]]. ([[WP:TW|TW]])","time":1382229037656,
  "source":"rc-pmtpa","channel":"#en.wikipedia"}
```

默认情况下，Hello Samza只读取维基百科的英语主题的文章。如果你觉得无趣，那就把应用停掉，并编辑wikipedia-feed.properties文件，使它能对不同语言的文章进行聚集计算。（在properties文件中只需要编辑一行，但为了照顾本书的格式，我们将其分成了多行。）

```
task.inputs=wikipedia.#en.wikipedia,
 wikipedia.#de.wikipedia,
 wikipedia.#fr.wikipedia,
 wikipedia.#pl.wikipedia,
 wikipedia.#ja.wikipedia,
 wikipedia.#it.wikipedia,
 wikipedia.#nl.wikipedia,
 wikipedia.#pt.wikipedia,
 wikipedia.#es.wikipedia,
 wikipedia.#ru.wikipedia,
 wikipedia.#sv.wikipedia,
 wikipedia.#zh.wikipedia,
 wikipedia.#fi.wikipedia
```

这样得到的文章流就有趣得多了，后面我们可以对这些文章进行分析。

Hello Samza应用还包含了一个解析器作业，它可以将原始的JSON数据转换为更有用的形式。由于Samza用Kafka进行通信，因此该作业可以

作为另外一个名为wikipedia-edits的Kafka主题。它的启动方式与第一个Samza作业相同，只是属性文件不同：

```
$ ./bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory \
> --config-path=file://`pwd`/config/wikipedia-parser.properties
```

该流的输出应该如下所示：


```
{"summary":"/* Episodes */ fix",
 "time":1382240999886,
 "title":"Cheers (season 5)",
 "flags":{
  "is-bot-edit":false,
  "is-talk":false,
  "is-unpatrolled":false,
  "is-new":false,
  "is-special":false,
  "is-minor":false
 },
"source":"rc-pmtpa",
```

```
"diff-url":"http://en.wikipedia.org/w/index.php?diff=577930400&oldid=
  577930256",
"diff-bytes":11,
"channel":"#en.wikipedia",
"unparsed-flags":"",
"user":"George Ho"
}

{"summary":"General Fixes using [[Project:AWB|AWB]]",
 "time":1382241001441,
 "title":"Degrassi: The Next Generation (season 5)",
 "flags":{
  "is-bot-edit":false,
  "is-talk":false,
  "is-unpatrolled":false,
  "is-new":false,"is-special":false,
  "is-minor":true
 },
"source":"rc-pmtpa",
"diff-url":"http://en.wikipedia.org/w/index.php?diff=577930401&oldid=
  577775744",
"diff-bytes":-4,
"channel":"#en.wikipedia",
"unparsed-flags":"M",
"user":"ChrisGualtieri"
}
```

由于Samza作业使用Kafka来传递流数据，如果你用其他系统（如Storm）来进行聚集计算，这个简单的入门项目也是可以用的。

8.1 定时计数与求和

计数与求和的最简单形式就是定时计数器。这时，事件的所有时间戳信息都会被忽略，就是简单按照到达顺序来处理事件。这种方式特别适合不需要可靠交付数据的采集机制，因此最适用于监控类应用。

定时计数器也非常适合Lambda架构。在Lambda架构中，有第二个进程负责纠正流处理环境中发生的任何数据丢失或重复计数。正常情况下，Kafka等系统输出的事件可能是无序的，但这通常无关紧要。另外，因为要重新读取数据而让事件重复出现的情况也比较少见。

在上面的两种情况下，都可以使用下面几节介绍的基本计数系统。这些系统都易于实现，并且开销也很小。不过，它们不适合需要很高准确度或需要进行幂等运算的系统。

8.1.1 基于Bolt的计数

Storm第一次发布时，并不包含任何基本运算。任何计数运算都是通过编写IRichBolt来实现的，由IRichBolt执行计数任务。为了能随着时间的推移不断产生输出，通常会用一个后台线程每隔固定的一段时间发送计数结果，然后重置本地的内存映射。在新版Storm（0.8.0之后的所有版本）中就不再需要后台线程了，因为现在Storm可以产生ticker事件。

实现基本的聚集计算bolt时，刚开始与实现任何其他bolt都很像。这里，bolt只有一个参数，它是向下一个bolt发送计数值之前要等待的秒

数：

```
public class EventCounterBolt implements IRichBolt {

  int                         updates= 10;

  public EventCounterBolt updateSeconds(int updates) {
this.updates = updates;return this;
}
  public int updateSeconds() { return updates; }
```

该bolt维护元素和元素计数值的临时映射，将其作为本地存储。与其他bolt类似，也要将采集器（collector）放入临时变量中。这两者都在准备阶段初始化。根据我们在前面代码中的配置，该bolt每隔几秒就输出计数值，因此必须声明一个输出流。最后，在getComponentConfiguration中配置该bolt的tick事件：

```
transient HashMap<String,Integer> counts;
transient OutputCollector       collector;

public void prepare(Map stormConf, TopologyContext context,
    OutputCollector collector) {
  this.counts     = new HashMap<String,Integer>();
  this.collector = collector;
}
public void declareOutputFields(OutputFieldsDeclarer declarer) {
declarer.declare(new Fields("timestamp","key","count"));
}

public Map<String, Object> getComponentConfiguration() {
Config conf = new Config();
  conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, updates);
  return conf;
}
```

注意这不是持久键-值存储，因此如果bolt任务由于某种原因终止，所有的临时数据都会丢失。

execute方法有两类输入元组。第一类元组是Storm根据前面的配置自

己发送的tick事件。为了检测这类事件，必须检查每个元组的源分量和流标识符：

```
public static boolean isTick(Tuple tuple) {
    return Constants.SYSTEM_COMPONENT_ID.equals(
      tuple.getSourceComponent()
    )
      && Constants.SYSTEM_TICK_STREAM_ID.equals(
      tuple.getSourceStreamId()
    );
}
```

下面的代码实现了execute方法。如果isTick方法返回true，那么bolt就会将目前存储在counts变量中的所有的值都发送出去。否则，它就将该值加1，并插入counts映射中。最后，要对该元组进行确认：

```
public void execute(Tuple input) {
    if(isTick(input)) {
      for(Entry<String, Integer> e : counts.entrySet()) {
        collector.emit(new Values(
          System.currentTimeMillis(),
          e.getKey(),
          e.getValue()
        ));
      }
      counts.clear();
    } else {
      String  key    = input.getString(0);
      Integer value  = counts.get(key);
      counts.put(key, 1 + (value == null ? 0 : value));
    }
    collector.ack(input);
  }
```

如果拓扑将要计数的项输出为流的第一个元素，那么就可以在拓扑中使用该bolt来计数。如果要用Redis等存储系统记录这些事件，那就用另一个bolt附加到该bolt的输出后面。

8.1.2 基于Trident的计数

相比用简单拓扑对事件计数，用Trident计数既有更容易的方面，又有更困难的方面。不再需要实现bolt来执行简单的计数和求和运算，而是实现Aggregator接口来处理事件计数任务。从这个意义上说，用Trident要简单一些。

Trident之所以使用聚集器，是因为它基本上是一个批量处理系统。聚集器以一个批次作为输入，再根据聚集函数的定义来生成一个输出。尽管多数时候编写自定义的聚集器是可行的，但对多数任务来说，使用内置的Count和Sum聚集器就足够了。下列代码使用第5章开发的拓扑的submitter接口，在Trident中实现了一个简单的事件计数器：

```
public StormTopology topology(String[] args) {
  TridentTopology topology = new TridentTopology();
topology.newStream("input", SimpleKafkaSpout.spout().configure(args))
       .aggregate(new Count(), new Fields("count"))
       .each(new Fields("count"), new PostFilter());
     ;
     return topology.build();
   }
```

PostFilter类很简单，用来与第7章中开发的node.js交付机制相衔接。它实现了与上一节中的IRichBolt计数器相同的功能，即将结果推入node.js应用，该应用默认监听3000号端口。

不同语言的维基百科文章事件

这里举一个稍复杂一点的例子。我们用Trident语言的groupBy特性，对特定语言的维基百科网站中的文章计数。与上一章介绍的不同，Trident不会生成tick事件，而是每处理完一个批次发送一个事件。每个批次默认情况下包含1000个事件。这样，前面介绍的简单拓扑就有点无趣了，因为它会始终输出[1000]。

要实现该拓扑，首先要使用来自第5章的JSONToTuple函数，从Samza提供的JSON输入流中抽取channel元素。然后使用groupBy操作符根据对应的channel将流分割成多个子流。最后，用聚集器对每种语言的文章计数。

要累积大约1000个事件，需要几分钟的时间，过一段时间后，类似于下面所示的事件就会被发送给node.js应用：

```
[ '#en.wikipedia', 598 ]
[ '#de.wikipedia', 43 ]
[ '#sv.wikipedia', 16 ]
[ '#fr.wikipedia', 61 ]
[ '#pt.wikipedia', 41 ]
[ '#es.wikipedia', 100 ]
[ '#zh.wikipedia', 27 ]
[ '#pl.wikipedia', 2 ]
[ '#nl.wikipedia', 9 ]
[ '#it.wikipedia', 28 ]
[ '#ru.wikipedia', 35 ]
[ '#fi.wikipedia', 1 ]
[ '#ja.wikipedia', 42 ]
```

8.1.3　基于Samza的计数

Samza用来实现计数作业的机制与Storm所用的tick事件非常相似。不过，它的实现要比Storm简洁一些。在Samza中，执行周期性任务的作业称为WindowedTasks，并且需要实现同名的接口。Samza不像Storm那样检查每个元组，而是只在相应的时候调用窗口事件。正因如此，Samza计数作业的完整代码很易读：

```
public class SimpleCountingJob
  implements WindowableTask, StreamTask, InitableTask {

  String       field = "field";
  SystemStream output;
  HashMap<String,Integer> counts = new HashMap<String,Integer>();

  public void process(IncomingMessageEnvelope msg,
    MessageCollector collector, TaskCoordinator coordinator)
    throws Exception {

    @SuppressWarnings("unchecked")
    Map<String,Object> obj = (Map<String,Object>)msg.getMessage();

    String key = obj.get(field).toString();
    Integer current = counts.get(key);
    counts.put(key, 1 + (current == null ? 0 : current));

  }

  public void window(MessageCollector collector,
    TaskCoordinator coordinator)
      throws Exception {
    collector.send(new OutgoingMessageEnvelope(output,counts));
    counts = new HashMap<String,Integer>();

  }

  public void init(Config config, TaskContext context) throws Exception
    {
    field  = config.get("count.field", "field");
    output = new SystemStream("kafka",config.get("count.output",
      "stats"));
  }

}
```

8.2　多分辨率时间序列的聚集计算

长期以来，系统监控类应用一直依赖所谓的轮循（round-robin）数据库来存储时间序列数据。这种数据库比较简单，会实现一个循环缓冲区来存储某些度量，例如每秒的CPU负载、事件数量等。

轮循数据库本质上就是循环缓冲区，它只能存储固定数量的数据点。因此，监控工具通常会按不断增大的时间间隔对每个度量维持多个这样的数据库。使用这种方法，就可以获得较短时间周期（例如24小时）内具有很高分辨率的数据（例如1秒的时间间隔）。对于单个度量，缓冲区需要86400个条目。下一个最大的缓冲区要在分钟级进行聚集计算。使用同样大小的缓冲区可以存放60天的分钟级数据。小时级的聚集计算则可以存储超过9年的数据。

量化框架

也可以不用固定大小的轮循数据库，而使用带有键过期策略的NoSQL后端存储，实现与轮循数据库相同的功能。这种方法尽管不够简洁，但它容易与前端服务集成，并且只要度量的数量保持稳定，这种方法占用的内存就是有界的。

多分辨率计数框架可以分为两部分。第一部分是一段通用的代码，用来定义聚集计算的时间范围和保留时间。第二部分针对特定的数据库实现定义聚集计算。尽管也可以让第二部分通用化，但如果这样做的话，或者让接口只实现最小的公共特性集合，或者要对没有实现某些特定性质的后

端数据库进行大量工作。

本节首先介绍将第一部分，即实现可以在各种环境下使用的通用类。然后针对第6章中的Redis键-值存储实现第二部分。之所以选择Redis，是因为它不但实现了很多操作，而且支持单后端数据库的键过期策略。

定义聚集器

Aggregator类的核心是Aggregate类。该类定义了聚集的分辨率以及保留时间。这个类本身很简单，主要包括一些便利函数的完整源代码。下面只给出了该类功能性部分的代码。类的第一部分定义各种聚集分辨率的格式化器。我们根据简单的毫秒时间值来选择使用哪种格式，因为毫秒时间比较易读，并且不会牺牲数字顺序和字典顺序：

```
public class Aggregate {

  private static final SimpleDateFormat millisecondFormatter
    = new SimpleDateFormat("yyyyMMddHHmmssSSS");
  private static final SimpleDateFormat secondFormatter
    = new SimpleDateFormat("yyyyMMddHHmmss");
  private static final SimpleDateFormat minuteFormatter

   = new SimpleDateFormat("yyyyMMddHHmm");
 private static final SimpleDateFormat hourFormatter
   = new SimpleDateFormat("yyyyMMddHH");
 private static final SimpleDateFormat dayFormatter
   = new SimpleDateFormat("yyyyMMdd");

 private static SimpleDateFormat formatForMillis(long millis) {
   if(millis < 1000)        return millisecondFormatter;
   if(millis < 60*1000)     return secondFormatter;
   if(millis < 3600*1000)   return minuteFormatter;
   if(millis < 86400*1000) return hourFormatter;
   return dayFormatter;
 }
```

接下来定义过期时间和保留时间。TimeUnit类是java.util.concurrent包中常被忽视的Java核心类。利用它可以在不同时间单位之间方便地转

换。这里，基本的时间格式是：

```
long     expirationMillis     = -1;
public Aggregate expire(long time,TimeUnit unit) {
  expirationMillis = TimeUnit.MILLISECONDS.convert(time, unit);
  return this;
}
public long expire(long time) {
  return expirationMillis == -1 ?
   -1 : expirationMillis*(time/expirationMillis);
}

long resolutionMillis = 1000;
TimeUnit unit = TimeUnit.SECONDS;
public Aggregate resolution(long time,TimeUnit unit) {
  resolutionMillis = TimeUnit.MILLISECONDS.convert(time, unit);
  this.unit = unit;
  return this;
}
```

量化函数（quantization function）以毫秒时间戳为输入，返回对应的量化时间戳。如果使用上述代码中的格式化字符串，它也可以返回一个适于在键-值存储系统中使用的时间键：

```
public TimeUnit quantizeUnit() { return unit; }

public long quantize(long time) {
  return resolutionMillis*(resolutionMillis/time);
}

public long quantize(long time,TimeUnit unit) {
  return quantize(
.meUnit.MILLISECONDS.convert(time, unit));
}

public String quantizeString(long time) {
  SimpleDateFormat sdf = formatForMillis(resolutionMillis);
  return sdf.format(new Date(time));
}
```

聚集驱动

定义聚集之后，需要一个驱动将其应用到特定函数。这是通过在

Aggregator类中实现each方法做到的。该方法计算每个量化值，然后执行命令：

```
public class Aggregator {

  ArrayList<Aggregate> aggregates = new ArrayList<Aggregate>();
  public Aggregator aggregate(Aggregate e) {
    aggregates.add(e);
    return this;
  }
  Expirer expirer = null;
  public Aggregator expirer(Expirer expirer) {
    this.expirer = expirer;
    return this;
  }

  public <E> void each(long timestamp,String key,
    E value,Command<E> cmd) {

    for(Aggregate a : aggregates) {
      long   quantized = a.quantize(timestamp);
      String quantizedString = a.quantizeString(timestamp);
      String expireKey = cmd.execute(timestamp,
        quantized,quantizedString,key, value);
      if(expireKey != null && expirer != null) {
        expirer.expire(expireKey, a.expire(timestamp));
      }
    }
  }

  public <E> void each(String key,E value,Command<E> cmd) {
    each(System.currentTimeMillis(),key,value,cmd);
  }


}
```

下一节我们会介绍Command＜E＞接口，它包含一个用来实现客户端操作的函数。Expirer接口与之类似，客户端用它实现过期特性，而这个特性是可选的。

注意　聚集驱动假定消息传递的时候是带有时间戳的。这样消息就可以乱序传递，即使有严重的延迟也没关系，但必须保证传递到后端存储的正确的“时间桶”。聚集驱动还实现简化版本的each方法，该方法只使用当前时间。

实现客户端

如果客户端使用聚集驱动，那么通常意味着要将一系列的Command<E>接口实现为匿名类。这些类被包装到一个方法中来实现方法的多分辨率版本。下列代码向控制台输出多分辨率的键和值：

```
public void doSomething(long timestamp,String key,String value) {
    aggregator.each(timestamp, key, value, new Command<String>() {

        public String execute(long timestamp, long quantized,
            String quantizedString, String key, String value) {
          String k = key+":"+quantizedString;
          System.out.println(k+"="+value);


         return k;
       }

    });
}
```

Command<E>接口只传递一个key和一个value。如果需要更多参数，可以将参数设为final，再将其传递进匿名类中。下面的doSomethingMore方法对此进行演示，它的第二个值参数就是这样做的：

```
public void doSomethingMore(long timestamp,String key,String value,
  final String another) {
    aggregator.each(timestamp, key, value, new Command<String>() {

        public String execute(long timestamp, long quantized,
            String quantizedString, String key, String value) {
          String k = key+":"+quantizedString;
          System.out.println(k+"="+value+","+another);
          return k;
        }

    });
}
```

多分辨率REDIS客户端

Redis特别适合多分辨率聚集计算。它的性能很高，数据类型很多，这使得它容易在较短时间内实现复杂的数据环境。在Redis的多分辨率聚

集计算客户端代码中，刚开始要定义聚集器和标准的Redis连接。下面的例子使用的是Redis的客户端，不过，任何Java Redis客户端都是可以的：

```
public class RedisClient implements Expirer {

  Jedis jedis;
  public RedisClient jedis(Jedis jedis) {
    this.jedis = jedis;
    return this;
  }

  Aggregator aggregator;
  public RedisClient aggregator(Aggregator aggregator) {
    this.aggregator = aggregator;
    aggregator.expirer(this);
    return this;
  }
```

Redis支持过期特性，因此该客户端实现了Expirer接口，用来支持聚集计算的过期特性（例如每5分钟的聚集两天后过期，每小时的聚集一周后过期，等等）。在该接口中有一个方法，对所有最高层的键都要调用这个方法：

```
public void expire(String key, long retainMillis) {
    jedis.expire(key, (int)(retainMillis/1000));
  }
```

实现多分辨率的getter方法没有什么用，但incrby、hincrby、zincrby和sadd这些主要的setter方法都是实现多分辨率更新操作不错的选择。例如，incrby是用简单的Command<E>来实现的：

```
public void incrBy(long timestamp,String key,long value) {
    aggregator.each(timestamp, key, value,new Command<Long>() {

        public String execute(long timestamp, long quantized,
            String quantizedString, String key, Long value) {
          String k = key+":"+quantizedString;
          jedis.incrBy(k, value);
          return k;
        }

    });
  }
```

像hincrby这样的更复杂的方法需要更多参数。这些参数被作为final参数传递给匿名命令函数：

```
public void hincrBy(long timestamp,String key,String field,
  final long by) {
    aggregator.each(timestamp, key, field, new Command<String>() {

        public String execute(long timestamp, long quantized,
            String quantizedString, String key, String value) {
          String k = key+":"+quantizedString;
          jedis.hincrBy(key, value, by);
          return k;
        }

    });
  }
```

在本章包含的代码中，剩下方法的实现方式与之相同。

8.3 随机优化

严格来讲，随机优化并不算是聚集计算，但确实有必要对其加以介绍。聚集计算和计数可以用来计算均值和标准差等简单值（下一章会详细介绍），还有一组类似的计算技术，这类技术统称为随机优化方法（stochastic optimization method）。

随机优化方法中最有名的是随机梯度下降，该技术广泛用于机器学习领域，用来处理超大的数据集。它的基本的思想是，存在某个函数F（B），该函数可以分解为函数f_i（B）的和，其中每个f_i（B）作用于一个数据点。我们的目标是找到使F（B）最小的B。在流（或在线）环境中，首先计算f_i（B）的梯度，然后从B的当前估计值中减去f_i（B）的梯度与r（r称为“学习速率”）的乘积，就可以得到B的下一个估计值。随机梯度下降本质上是一种特殊的求和运算，可以用来计算许多有趣的值。

有一项称为随机平均的类似技术，也常常以同样的方式计算数据流的中位数。中位数是数据集中的某个值，如果对该数据集排序，其中一半的数据要比该值小，另一半的数据比该值大。在数据流上计算中位数通常比较难，因为需要采集全部数据并排序。

使用随机优化计算中位数的近似值时，我们关注的是中位数M的当前估计值。如果在流中观察到的下一个值比M大，那就将M加上r。如果比M小，那就将M减去r。当M接近于中位数时，对它的增加和减少操作趋向平衡，因此M的取值就稳定了：

```java
public class MedianEstimator {

  double rate = 1.0;
  double current = Double.POSITIVE_INFINITY;

  public MedianEstimator(double rate) {
    this.rate = rate;
  }

  public double update(double value) {
    if(current == Double.POSITIVE_INFINITY)
      current = value;
    if(current == value) return current;
    current = current + (current < value ? rate : -rate);
    return current;
  }

}
```

后面的章节会用到这项技术来学习更加有趣的量，例如预测分类器的参数。

8.4 时间序列数据的交付

在第7章中，我们深入讨论了如何将数据从后端系统移动到基于Web的客户端进行渲染。对于时间序列数据也是同样做法，如果后端支持某种发布-订阅机制，那就更容易。当时间序列聚集器发送数据时，它可以直接将其发送到channel中，或者更新后端存储，并向后端存储channel发送通知。在这两种方式中，后者更为常见。

向REDIS客户端发送通知

向上一节的Redis客户端加入发布订阅通知是很容易的。首先，Aggregator类需要有调用通知事件的途径。这是通过Notifier接口来完成的：

```
public interface Notifier {
  public void notify(String key,String resolution,
    long timestamp,long quantized,String quantizedString);
}
```

每当each方法中更新时，该接口就会被调用。当Notifier接口中有通知时，下列代码根据情况来通知channel：

```
if(notifier != null) {
  notifier.notify(key,a.resolutionString(),timestamp,
    quantized,quantizedString);
}
```

然后Redis客户端将更新发布到相应的channel，这样数据消费者就可以得到更新。这里，更新的分辨率包含在channel中，这样消费者就可以订阅相应的分辨率：

```
public void notify(String key,String resolution, long timestamp,
  long quantized, String quantizedString) {

jedis.publish(key+":"+resolution, quantizedString);
}
```

如果在开发环境中没有实时数据源，另一种选择是使用简单的JavaScript驱动发送“实时”事件，从而模拟实时数据流。该驱动以正弦波的形式每秒发送消息，这样就可以测试本节后面描述的可视化方法：

```
(function($) {
  var incr = 1/(2*Math.PI);
  var x = 0;
  var t = 0;
  var i = setInterval(function() {
    var y = 100*Math.sin(x);
    var $d = $(document);
    $d.trigger('data:signed',y);
    $d.trigger('data:unsigned',100+y);
    x += incr;
  },100);

})(jQuery);
```

8.4.1　用D3.js绘制带状图

带状（strip）图用x轴表示时间，用y轴表示度量。例如，图8-1就是一个正弦数据的简单带状图。

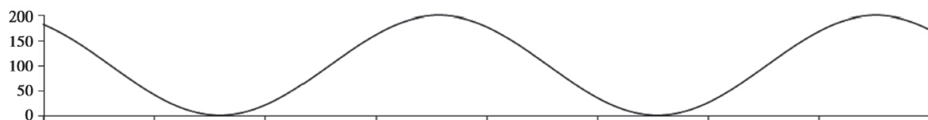


图　8-1

忽略掉上一章讲到的启动D3的步骤，我们首先要做的就是，对带状图

定义要显示的x和y的刻度以及坐标轴：

```
var y = d3.scale.linear().domain([0,200]).range([height,0]);
var x = d3.scale.linear().domain([0,size]).range([0,width]);
x.axis = d3.svg.axis().scale(x).orient("bottom");

var axis = g.append("g")
.attr("class","x axis")
.attr("transform","translate(0,"+height+")")
.call(x.axis);

g.append("g")
      .attr("class", "y axis")
      .call(d3.svg.axis().scale(y).ticks(5).orient("left"));
```

接下来，绘制曲线来表示data变量中的时间序列：

```
var line = d3.svg.line().interpolate("basis")
  .x(function(d,i) { return x(i); })

   .y(function(d,i) { return y(d.y); });

 var path =
   g.append("g").append("path").data([data]).attr("class","line");
```

每当有事件到达，data变量首先被更新。然后就会重新绘制曲线。最后，删除data的第一个元素。首先要重绘曲线是因为改变第一个元素会修改曲线的控制点。通过将第一个元素隐藏到y轴的后面，用户就不会觉察到这个变化：

```
function update() {
  g.select(".line").attr("d",line);
}

var i = 0;
$(document).on('data:unsigned',function(event,y) {
  data.push({x:i,y:y});
  update();
  if(data.length > size+2) data.shift();
  i++;

});
```

8.4.2　高速Canvas图

当时间序列的实时更新速度快，但不算很快时，迄今为止我们使用的方法都还奏效。对于特定度量进行更新的常见时间间隔是10秒到1分钟。在大型系统中，如果更新速度为每秒1次或者更快，则容易出现问题，因为可伸缩矢量图形（SVG）的渲染性能有限。

除了使用SVG，另一种方法是渲染Canvas元素。Canvas元素的渲染通常要比SVG快一些，但它的样式不灵活。因此，仍然可以用标准SVG元素来渲染坐标轴等静态元素，并制定样式，然后将SVG元素置于Canvas元素之上。

即使用上面这种方法也有局限性，导致无法真正地快速更新，因为每一帧都必须渲染整个canvas。对于非常简单的图形来说，高速更新机制可以通过只更新canvas最左边的部分来提高渲染速度。为此，首先要初始化要绘制的canvas元素。这里的canvas在HTML文档中名为canvas1：

```
var size   = 80,width  = 960,height = 120;
var canvas = document.getElementById('canvas1');
var ctx    = canvas.getContext('2d');

var w = width/size;
var y = d3.scale.linear().domain([0,200]).range([2,height-4]);
```

由于线段比较短，绘制带状图最简单的方法是从前面的点到当前点使用线性插值。这使得更新步骤变得十分简单。带状图的上下文只绘制从点（width–w，lastY）到点（width，newY）之间的直线，从而生成一个线段：

```
var lastY;

$(document).on('data:unsigned',function(event,newY) {

if(lastY != undefined) {

    shiftCanvasLeft();
    ctx.beginPath();
    ctx.moveTo(width-w,y(lastY));
    ctx.lineTo(width,y(newY));
    ctx.stroke();
  }
  lastY = newY;
});
```

注意，在绘制线段之前，要调用shiftCanvasLeft函数。该函数首先获取当前canvas的一个快照，然后左偏w个像素对其重新绘制：

```
function shiftCanvasLeft() {
    var img = ctx.getImageData(0,0,width,height);
    ctx.clearRect(0,0,width,height);
    ctx.putImageData(img,-w,0);
  }
```

对于绘制高性能的带状图来说，再没有什么是必须要做的了。该方法还有一个很好的性质，它不需要存储数据本身，只需要存储最后一个值。如果使用条形图而不是带状图，那甚至连最后一个值也不用存储。

Hummingbird

使用该方法的首个实时应用就是Hummingbird，它是由Gilt Group工程师们开发的。该应用是自包含的，它用网站的1×1像素来记录一次“命中”。这些命中数据存储在MongoDB中。然后，利用第7章的socket.io库，用node.js应用通过WebSocket交付数据。

8.4.3　地平线图

带状图的一个近亲是地平线图，它是由Jeffrey Heer、Nicholas Kong和Maneesh Agrawala提出的。地平线图的思想是要能可视化大量相关联的时间序列变量。如果相关联的时间序列不多，那用带状图就可以很容易地做到。但是由于垂直空间比较扁，随着变量数量的增加，检测每个带状图的变化就越来越困难。

为了解决这个问题，地平线图使用区域来取代线条，并且将y轴上数据值卷折起来。为了避免丢失较高的值，使用密度来表示区域的重叠。典型的卷折因子是3，如图8-2所示，这样图形只需要原来的三分之一甚至更少的面积，却仍然能让你看到图形的细节结构。



图 8-2

创建图8-2中图形的代码实际上与渲染带状图的代码非常相似。只不过不是渲染一个线条，而是渲染三个重叠区域。该函数用相应的值域集合定义每个区域，利用D3的clamp选项来限制输出范围，使其与输入值的大小无关：

```
function region(min,max) {
  var y     = d3.scale.linear()
              .domain([min,max])
              .range([height,0])
```

```
              .clamp(true);
  var area = d3.svg.area()
  .x(function(d,i) { return x(i); })
  .y0(height)
  .y1(function(d,i) { return y(d.y); });
  var path = g.append("g")
              .append("path")
              .data([data])
              .attr("class","horizon");
  return function() {
    path.attr("d",area);
  }
}
```

注意，每个区域实际上都维护一个指向整个数据的指针，因此不需要进行额外的数据维护工作。接下来的函数简单地将值域划分为多个区域。用对应值域上的三个区域来调用该函数，就生成了地平线图：

```
function make(min,max,split) {
  var regions = [];
  var last    = min;
  var incr    = (max-min)/split;


  for(var i=0;i<split;i++) {
    var to = last+incr;
    regions.push(region(last,to));
    last = to;
  }

  return regions;
}
var regions = make;
```

Region函数会返回一个更新函数，因此更新函数只是依次调用每个区域的更新函数来生成图8-2中的图形：

```
function update() {
  regions.forEach(function(r) { r(); });
}
```

Cubism.js

从上面的介绍可以看出，使用D3.js来生成简单的地平线图并不难。另一个选择是利用cubism.js项目。该项目是由Mike Bostock编写的D3.js插件。Mike Bostock是D3最初的开发人员之一，也是该项目活跃的维护者。他在Square公司的时候，为内部项目的时间序列可视化编写了cubism.js，后来就把它开源了出来。

除了实现地平线图的渲染功能，cubism.js还有内置的数据连接器。它对Graphite和Cube的支持是现成的，另外，它的库也支持来自其他资源的插件。

8.5 总结

本章基于前面几章的基础架构，建立了轻量级的聚集计算框架。我们重点介绍了时间序列数据框架，因为这往往是实时流的应用领域。

当然，时间序列数据的可视化不限于带状图及与其相近的图形。随着高性能Web浏览器的出现，完全可以用颜色和动画来创建令人眼花缭乱的可视化。不过，对于流数据来说，图形是否“一目了然”是一个需要注意的因素。如果不与时间分量直接结合，可视化就很难达到一目了然的效果。如果目标是吸引用户来观看数据的展示，可能就有必要在可视化中加入大量动画元素。

多数时候，我们的目标是尽可能快地提供尽量多的信息。这种情况下，如果用带状图和地平线图这样“无聊”的可视化，反倒能以易于接受的形式传递信息。

第9章　流数据的统计近似

数据流的许多基本属性都可以通过基本的计数方法来获得。和、平均值、最小值、最大值以及其他顺序统计量（在某种程度上）都能以O（1）的更新时间和O（1）的存储空间计算出来。多数系统满足于这些基本的统计值，因为计算更复杂的值要求低延迟和近乎无限的存储空间，代价过于高昂。

第9章和第10章从统计学的角度解决这个问题。统计学原本是用来解决全国人口普查时面临的高昂成本或耗时这些问题的。后来统计学开发了工具包支持使用抽样来对人口进行推断，这要用到概率论提供的工具包。本章对统计学方法和概念提供了简要介绍，包括概率和统计的基础知识，利用这些知识可以回答关于数据的问题，而不是只列出表格式的结果。

本章中的技术与流数据分析并不是特别相关。它们适用于有限数据集，与数据集的大小无关。当然，也可以将其稍加修改来应用于数据流。本章后面会讨论从数据流中进行高效抽样的方法。可以对这些抽样进行统计分析来进行深入分析和预测等。

概率框架对于维护流数据概要的专门数据结构的开发也是有用的。第10章深入讨论这些专门数据结构，这需要对随机值的行为有一定的理解，本章剩下的部分会讨论这一点。

9.1　数值计算库

后面几章大量用到数值计算函数，这在多数语言的标准数学库中是找不到的。本章中的例子多数是用Java编写的，使用开源的Colt数值计算库，该库是由欧洲核能研究组织（European Organization for Nuclear Research，CERN）开发的，该组织也是发明Web的组织。Colt使用GUN Lesser General Public License（LGPL）许可证，并明确说明了禁止用于军事应用。该库的Maven依赖如下所示：

```
<dependency>
  <groupId>colt</groupId>
  <artifactId>colt</artifactId>
  <version>1.2.0</version>
</dependency>
```

Apache公共库也为数值计算提供了Java库。

其他语言也有很好的数值计算库，它们通常提供接口来使用经过完备测试的高性能Fortran库。对于商业应用来说，长期以来的主流是NAG库和IMSL（International Mathematics and Statistics Library），这两者有针对多种语言的库。

除了本书中的Colt库之外，C语言用户也可以选择GUN Science Library（http://www.gun.org/software/gsl/ ）这样的开源库。还可以选择R统计语言（http://r-project.org）的嵌入式版本，该版本开放了其许多核心数值计算例程。对于C＋＋用户，Boost库（http://boost.org ）是主流选择，它支持许多应用。

对于高层语言来说，可供选择的库要少一些。Python的NumPy（http://numpy.org ）和SciPy（http://scipy.org ）可能是最成熟的数值计算库。Ruby和JavaScript等其他语言有专用库，但始终没有整合成一个综合的数值计算库。

9.2 概率和分布

概率论是整个统计学领域的基础。概率论原本用来理解游戏和博彩，它的经典应用是对装满了彩色小球的一个或多个罐子研究下列问题："从罐子中取出的下一个小球是红色的概率是多少？"当然，这个问题的答案是罐中红色球的数量除以罐中球的总数，即：

P（ball = red） = #red balls/#balls

概率的总和始终为1，因此也可以得到相反事件的概率："下一个球不是红色的概率是多少？"例如：

P（ball ≠ red） = 1 - P（ball = red）

事件的组合也具有类似的形式。取出一个红色球或一个白色球的概率为：

P（ball = red or ball = white） = （#red balls + #white balls）/ #balls

这个概率等于P（ball = red） + P（ball = white）。实际上，只要事件是相互独立的，它们都可以这样相加。

处理"与"事件时，问题会稍复杂一些。例如"从罐中先取出红球后取出白球的概率是多少？"通常要将这些事件的概率相乘而不是相加。因此，要回答这个问题，取出红球的概率和取出白球的概率都是需要计算的。取出红球的概率仍然是原来的值。但接着取出白球的概率则取决于取红球事

件。

条件概率（conditional probability）记为P（A|B），描述的是第一个事件和第二个事件之间的关系。它读作“假定B已经发生时A发生的概率。”这样就可以将前面提到的“与”事件分解为更易处理的形式：

P（A and B）=P（A|B）P（B）=P（B|A）P（A）

因此，如果将红球取出之后又放回罐中，上面的公式将变成：

P（ball 1 = red and ball 2 = white）=P（ball = red）P（ball = white|ball = red）

=P（ball = red）P（ball = white）=（#red/#balls）×（#white/#balls）

如果没有将红球放回罐中，那罐中就会少一个球，概率就应该是：

P（ball 1 = red and ball 2 = white）=（#red/#balls）×（#white/#balls - 1）

还可以将该公式扩展来处理更复杂的离散事件，例如如果总共取了Y个球，其中有X个球是红球的概率。例如，如果X为2，Y为5，那么蛮力方法就是将取出两个红球的所有方式穷举出来。如果用R表示红球，用N表示非红球（具体什么颜色无所谓），那取5个球中有2个红球就有10种可能的方式：

·RRNNN

·RNRNN

·RNNRN

·RNNNR

·NRRNN

·NRNRN

·NRNNR

·NNRRN

·NNRNR

·NNNRR

如果每次取出球之后又放回，取出一个红球的概率就是#red/#balls，取出另外颜色的球的概率就是1 - #red/#balls。相乘之后得到（#red/#balls）^{2} （1 - #red/#balls）^{3} ，一般化的公式为（#red/#balls）^{i} （1 - #red/#balls）$^{n-i}$ 。

接下来唯一要做的就是考虑5个球的10种不同顺序。首先将每个球分别编号，然后就可以查看这些带编号小球的排序方式的数量。总共有5×4×3×2×1 = 120种方式。这通常用5！来表示，5！称为阶乘函数（factorial function）。但是，这里没有区分红球和非红球，因此该方法会导致重复计数。同理，两个红球有2！种排序方式，3个非红球有3！种排序方式。每个最终的10种排序实际上都被重复计数了2！×3！遍，因此

两者相除得到5！/（2！×3！）=10种排序。换句话说，有“5选2”种可能的组合。一般化的表示为“n选k”，即从n个对象中选出k个对象的不同方式的数量。可能的组合的数量可以由公式n！/（k！×（n - k）！）表示。

9.2.1　期望和方差

如果取5个球的实验重复无数次，取出红球的平均数量是多少？统计学家会询问“红球的期望数量是多少？”，它们将这个问题的答案称作期望（expectation）。在统计学术语中，通常将其记作E[X]（使用中括号），其中X是随机变量（random variable）。

期望的定义非常简单。对于上面这样的例子来说，即包含取两个球这样的离散事件，期望就是X的每个可能值乘以X取该值的概率，如下所示：

$$0\times P(X=0)+1\times P(X=1)+\ldots+n\times P(X=n)$$

更一般地说，实际上可以对X的任何函数定义期望，记作E[f（x）]，其计算方式相同：

$$f(0)\times P(X=0)+f(1)\times P(X=1)+\ldots+f(n)\times P(X=n)$$

9.2.1.1　均值和方差

多数人了解期望E[X]，将其称为X的均值（mean或average）。使用上一节中的例子，很容易算出从5个球中取出红球的期望数量（或均值）。取出0个红球的情况可以忽略掉，因为0乘以任何值都等于0。这样就只需要将剩下的4种情况加在一起：

$1\times5\times p\times(1-p)^4+2\times10\times p^2\times(1-p)^3+3\times10\times p^3\times(1-p)^2+4\times5\times p^4\times(1-p)+5\times1\times p^5$

计算出所有项的乘积，然后进行烦琐的销项，最终会得到取5个球中红球的期望数量是5p。

当然，从某组特定的5个球中取出红球的实际数量是变化无常的。根据p的值，期望值甚至可能不是整数，因此取出“期望”数量的红球是不可能的。观察到的红球数量在红球期望数量两侧的分散程度是由方差（variance）来衡量的，记作Var（X）。多数人曾见到过方差的平方根，称作标准差（standard deviation）。

方差其实也是一种期望。方差的定义是$\mathrm{Var}(X)=E[(X-E(X))^2]$。它可以展开为$E[X^2]-(E[X])^2$。它的计算方法与任何其他期望相同。对前面的例子来说，我们忽略掉中间计算过程，Var（X）应为$5\times p\times(1-p)$。

9.2.1.2 其他矩

均值和方差专指X二次幂（在方差或其他更高次幂的情况下是X—E[X]）的期望，除此之外，X的一些其他幂也有专门的名称。总的来说，X期望的这些幂称为矩（moment），如果它们是X - E[X]的幂，则称为X的中心矩（central moment）。

本章后面几节会用这些矩来帮助我们从观察到的数据中计算有趣的参数。例如，在前几节使用的例子中，p是一个循环参数（recurring parameter）。在所提出的问题中，p等于罐中红球的数量除上球的总数。

当p已知时，很容易计算出结果。但是，多数情况下我们能观察到X的一些取值，因此主要任务就是求出p。

9.2.2 统计分布

给定X的一些观察值（这些观察值称为数据（data）），确定p的可能取值范围需要使用称为分布（distribution）的数据模型。顾名思义，分布描述的是概率在X的所有可能取值上的分布方式。

也有一些例外，例如非参数方法，但几乎所有类似的计算都基于对数据底层的特定数学模型做出的显式或隐式的声明。这些底层模型都非常有名，因此它们都被命名并有标准解释。下面两节介绍这些著名模型中的一些成员，它们可以分为两类：离散分布和连续分布。

9.2.3 离散分布

离散分布由概率质量函数（Probability Mass Function，PMF）描述，该函数是随机变量X取特定值k的概率，通常记为p（k）。将k所有可能的取值对应的p（k）加在一起，总和始终为1。有许多离散分布，它们都有已充分研究过的解释，但对于本书的读者来说，下面三节介绍的五个分布是特别有用的。

9.2.3.1 二项分布和超几何分布

上一节中的概率分布统称二项分布（binominal distribution）。二项分布的概率质量函数的实现如下：

```
public static double dbinom(long k,double n,double p) {
  return Arithmetic.binomial(n, k)
*Math.pow(p, k)
*Math.pow(1-p, n-k);
}
```

其中来自Colt数值计算库的Arithmetic.binomial函数实现了上一节中的“n选k”的组合函数。这通常称为二项式系数（binomial coefficient）。

除了描述取n个球然后放回，有i个特定颜色球的概率之外，该分布还用来对概率不会因实验次数而变化的其他过程建模。其中包括抛n次硬币出现正面的数量，或者每次摇骰子多个骰子点数的总和。一般来说，该概率为n次实验有i次成功的概率，其中成功的概率为p。

如果每次取球后不将球放回，那就是超几何分布（hypergeometric distribution）。如果K是红球的总数，N是球的总数，则其分布函数的实现如下：

```
public static double dhypergeom(long k,long n,long N,long K) {
  return (Arithmetic.binomial(K, k)
*Arithmetic.binomial(N-K, n-k))
  /Arithmetic.binomial(N, n);
}
```

9.2.3.2　几何分布和负二项分布

如果每次取球都放回，取到第一个红球前已取的非红球的数量的分布称为几何分布（geometric distribution，有时也称作首次成功分布（first success distribution））：

```
public static double dgeometric(long i,double p) {
    return Math.pow(1-p, i-1)*p;
  }
```

将该分布扩展成取到第r个红球前已取球的数量（其中每次取球都放回），新的分布称为负二项分布（negative binomial distribution）。几何分布实际上是负二项分布的特例（对应r = 1的情况），因此不出所料它们的实现也类似：

```
public static double dnegbinom(long i,long r,double p) {
    return Arithmetic.binomial(i-1, i-r)*Math.pow(1-p, r)*Math.pow(p, i-r);
}
```

这些分布有许多应用。从上面的介绍你或许能猜到它们可以用于质量控制和监控类应用中。

9.2.3.3 泊松分布

本节介绍的最后一个分布是泊松分布（Poisson distribution）。该分布对有限时间范围内发生的事件数量的分布进行建模。例如，每小时接到的电话数量，或者等待红灯的汽车数量：

```
public static double dpois(int i,double p) {
    return Math.pow(p, i)*Math.exp(-p)
/Arithmetic.factorial(i);
}
```

当p保持不变而n非常大时，泊松分布还可以用来近似表示二项分布。这时，使用泊松分布往往要比使用二项分布更方便。当$n \geq 100$且$n \times p \leq 10$时，可以将泊松分布视为二项分布式的一个很好的近似。

9.2.4 连续分布

当分布要表示的是连续数据时，例如人群中身高的分布数据，事情就

有点不同了。与离散的情况不同，这时难以得到P（X = i）。当有人说他们有6英尺 [1] 高时，这实际上并不是准确值。他们的身高可能是6英尺加上某个误差量化因子（例如半英寸）。

为了解决这个问题，连续分布的定义不是P（X = i）的形式，而是P（X≤x），该函数称为累积分布函数（Cumulative Distribution Function，CDF）。为了让函数看起来更像是连续分布情形，取P（X≤x ＋h）- P（X≤x），并令h无穷小。你可能会说这不就是计算导数吗，你说的没错。该函数通常记为p（x），称为概率密度函数（probability dernity function），就是CDF的导数。（离散的版本称为概率质量函数。）

9.2.4.1　正态分布和卡方分布

正态分布（normal distribution）也称为高斯分布（Gaussian distribution），大概是所有统计分布中最著名的。其中一个原因是它的函数形式可以对许多不同的过程产生很好的结果。例如，聚类算法经常明确假设聚类服从正态分布。

正态分布之所以著名，另外一个更重要的原因是，随着实验次数趋向无穷，随机变量观测值均值的分布会收敛为正态分布。令人惊讶的是，只要符合特定条件（通常都是符合的），不论随机变量服从何种分布，该性质都成立。换句话说，如果有足够的数据，那你就几乎可以用正态分布近似任何数据（甚至是离散分布！）。

正态分布有两个参数：均值（mu）和标准差（sigma）。正态分布的

概率密度函数的实现很简单，其中x可以是任何实数：

```
public static double dnorm(double x,double mu,double sig) {
     return Math.exp(
       Math.pow(x-mu, 2)
       /Math.sqrt(2*sig*sig)
     )/Math.sqrt(2*Math.PI*sig*sig);
   }
```

如果X_1 ，…，X_k 服从正态分布，那么它们的平方和就服从自由度为k的卡方分布。因此单个正态分布的随机变量的平方就服从自由度为1的卡方分布。卡方分布用来对正态分布的方差进行建模，也用来分析“列联表”（contingency table），列联表可以用来确定事件在两个组中的出现频率是否不同。该分布的概率密度函数很复杂：

```
public static double dchisq(double x,double k) {
   return (Math.pow(x,k/2.0 - 1.0)
*Math.exp(-x/2.0))
/(Math.pow(2,k/2.0)*Gamma.gamma(k/2.0));
}
```

上述概率密度函数中的Gamma.gamma（）函数实质上是阶乘分布的连续版本。实际上，Arithmetic.factorial（n）等于Gamma.gamma（n - 1）。

9.2.4.2 指数分布、gamma分布和beta分布

泊松分布是给定时间范围内事件发生的数量，指数分布 模型则是这些事件之间的等待时间。在对队列建模时，指数分布常与泊松分布一起使用。指数分布只有一个参数，它的概率密度函数很简单：

```
public static double dexp(double x,double p) {
     return p*Math.exp(-x*p);
   }
```

当每个事件等待时间服从指数分布时，从第一个事件直到第k个事件的等待时间的分布就是gamma分布 。该分布有两个参数：p和k。p称为频率（rate），来自于指数分布。第二个参数k称为形状（shape），它表示事件的数量。该分布的概率密度函数清楚地给出两种分布之间的关系：

```
public static double dgamma(double x,double k,double p) {
    return Math.pow(p,k)*Math.pow(x,k-1)*Math.exp(-p*x)/Gamma.gamma(k);
}
```

这种关系与几何分布和负二项分布之间的关系很相似。实质上，指数分布就是gamma分布的特例。卡方分布也是gamma分布的特例，其中形状参数k是自由度的二分之一，频率参数p是1/2。

注意　除了是k个指数分布的随机变量的和之外，gamma分布的随机变量还可以表示为X~Gamma（a，b）= Y/b，其中Y~Gamma（a，1）。该性质常在处理gamma随机变量时使用，用来简化密度函数。

如果两个变量X和Y都服从gamma分布，并且参数k的取值相同，而参数p的取值不同，那么X/（X + Y）就服从beta分布 。beta分布用来建模取值为0~1之间的随机变量，因此常用来对频率建模。

[0，1]上的均匀分布是beta分布的特例，这时a = 1，k = 1。

9.2.5　联合分布

迄今为止，所有的讨论都集中于单个分布。单个分布对于考察单个变量有用，但多数问题感兴趣的是理解两个或多个随机变量之间的关系。在统计学中，是通过联合分布 （joint distribution）来定义变量之间的关系

的。

联合分布包含两个或多个随机变量，并像所有其他分布一样拥有概率密度函数和累积密度函数。联合分布对于变量是否具有相同的“类型”没有限制，并且既包含离散变量又包含连续变量的情况很常见。

如果两个变量是独立的，它们的联合分布就很简单。其概率密度函数就是两者的乘积。如果一个或多个变量条件依赖于另外的变量，要构造联合分布的概率密度函数，就要使用与定义条件概率相同的机制。

例如，考虑一个人群，假设随机选择一人是女性的概率是p，男性和女性子群都服从正态分布，其中男性子群的正态分布均值为hm，女性为hf，标准差均为s。f（height，gender）的联合概率密度函数的实现如下：

```
public static double dnormGender(double h,double gender,
double hm,double hf,double sig,double p) {
  return Math.pow(p,gender==1 ? 1 : 0)
*Math.pow(1-p,gender==1 ? 0 : 1)
*dnorm(h,hm + (gender == 1 ? hf-hm : 0),sig);
}
```

协方差和相关系数

两个随机变量的协方差是对这两个随机变量之间关联的密切程度的度量。只要定义了两个随机变量的二阶矩（方差），那协方差就可以简单地表示为E[（X - E[X]）×（Y - E[Y]）]。这简化了教科书中常用的形式：E[XY] - E[X]E[Y]。据此你可以看出方差就是变量与自身密切程度的度量，将X替换为Y就得到前面提到的常见的方差公式。

两个随机变量之间的相关系数是协方差的归一化形式：

ρ（x，y）＝Cov（X，Y）/Var（X）×Var（Y）

这确保了相关系数的值处于-1～1之间。注意，尽管独立随机变量之间的相关系数为0，但相关系数为0的两个随机变量并不一定是相互独立的。

[1] 1英尺＝0.3048米。

9.3 参数估计

假定某些观察数据服从特定分布之后，现在可以推断一个或多个未知参数的可能值。也可以推断参数可能值的范围，或者根据给定的来自两个不同观察数据的集合（可能来自两个不同组）来推断这些估计参数是否可能真的不同。

本节剩下的内容重点讨论对未知分布的参数进行推断的通用方法。对于知名分布，我们给出对给定数据进行参数估计的封闭解表达式。

9.3.1 参数推断

如果x_1，...，x_n是对服从F（x）分布的随机变量的观察值，那就用著名的“极大似然估计”或简称为“极大似然”）推断出F（x）的参数。顾名思义，该过程求使分布的似然函数值最大的参数值。

似然函数就是每个观察值的概率密度函数的乘积：

$$l(x_1, x_2, \ldots, x_n) = f(x_1) \times f(x_2) \times \ldots f(x_n)$$

我们通常令似然函数的负对数最小化，来求出使似然函数值最大的参数值。计算似然函数的负对数的导数，使该结果函数等于0，然后求出令该结果函数最小的参数。如果分布有多个参数，那就使用偏导数。例如，可以使用前面介绍的二项分布的例子，用每取5个球观察到的红球的数量来估计罐中红球的数量。似然函数负对数的导数用如下表达式表示：

$$-(x_1+\ldots+x_k)/p-(k\times5-(x_1+\ldots+x_k))/(1-p)=0$$

二项分布的常数项没有列在导数中，因为常数项与p无关，所以当对p求导时它会被从等式中消掉。根据上述公式可以求得p为（$x_1+\ldots+x_k$）/5×k，其中x_i是每取5个球观察到的红球的数量。

对知名分布的参数通常不需要计算极大似然估计。如果它们的极大似然估计存在封闭解，那么通常都是众所周知的，可以在各种文献中找到。

另一种估计分布参数的“蛮力”技术称为“矩方法”（method of moment）。该方法将分布的矩$E[X^k]$表示为未知参数的函数，以此构成方程组。因此，如果分布有两个未知参数，矩方法就是用分布的前两个矩来构成二元方程组。抽样矩的观察值是很容易计算的。将计算结果代入方程中，就可以求出方程组的解。

例如，gamma分布有形状和尺度（scale）两个参数。要用矩方法来估计这些参数值，就需要前两个矩：$m_1=k\times p$和$m_2=p^2\times k\times(k+1)$。给定一个样本，假设该样本取自服从gamma分布的数据，那么前两个矩分别是数据的均值和数据平方的均值。将其代入m_1和m_2，就可以求出p和k。

但是，矩函数从何而来？比较费劲的方式是对相应的矩进行手工积分，就像期望和方差的定义中介绍的那样。不过，多数知名分布都有所谓的矩母函数（moment generating function）。该函数的定义是$E(e^{sX})$，对其多次求导就会生成矩函数。如果需要二阶矩，那就求导两次；需要5阶矩，就求导5次，依此类推。然后令s=0，就可以得到矩函数。前面

介绍的gamma分布中使用的函数就是这样得来的。总的来说，这些母函数，以及可以用于计算随机变量的随机和等问题的其他形式的母函数，都可以在表格以及维基百科、Wolfram Alpha搜索引擎等其他资源中找到。

9.3.2 Delta方法

矩方法（method of moments）实质上提供了一个工具来计算随机变量X的幂的期望。然后使用X的幂来求分布参数的估计值。其实还有更好的方法。对随机变量X的几乎任意的函数，可以使用Delta方法 这个工具求得该函数期望的近似值。

使用Delta方法，随机变量的近似期望和近似方差可以表示为：

$E[f(X)]=f(E[X])$

$Var(f(x))=f'(E[X])^2\times Var(X)$

$f''(a)$是函数$f(x)$的二阶导数。下列近似公式是使用泰勒展开式得到的，泰勒展开式往往在应用数学中用来对复杂的非线性函数进行近似。如果要估计某个点“a”，泰勒展开式提供的公式可以将函数展开成多个部分。例如，下面就是$f(x)$的包含线性项和二次项的泰勒展开式，也即二阶展开式：

$f(x)=f(a)+(x-a)f'(a)+(x-a)^2f''(a)/2+e$

泰勒展开式剩下的部分是误差项“e”，包含了3阶和更高阶的展开项。这可以用来计算近似值的误差，但就本书的目的而言，该误差可以被视

为“足够小”。

要推导Delta方法的结果，随机变量X的函数f（X）关于X的期望E[X]展开为：

$$f(X)=f(E[X])+(X-E[X])f'(E[X])+(X-E[X])^2 f''(E[X])/2+e$$

取该展开式的期望，从而得到E[f（X）]的二阶近似：

$$E[f(X)]=E[f(E[X])+(X-E[X])f'(E[X])+(X-E[X])^2 f''(E[X])/2+e]$$

$$=E[f(E[X])]+E[(X-E[X])f'(E[X])]+E[(X-E[X])^2 f''(E[X])/2]$$

这里，E[X]是常数，因此f（E[X]）也是常数。常数的期望还是常数，$E[b]=b$。$b\times X$的期望等于X的期望乘上常数b，即$E[b\times X]=b\times E[X]$。将其应用到上述公式可以得到：

$$E[f(X)]=f(E[X])+(E[X]-E[X])\times f'(E[X])+Var(X)f''(E[X])/2+e$$

$$=f(E[X])+Var(X)f''(E[X])/2+e$$

如果只取最终公式的第一项，那就得到对上述期望的Delta方法一阶近似。如果将随机变量X的方差和函数的二阶导数包含进来，那就得到二阶近似。该公式还说明，一阶近似的误差取决于随机变量X的方差。

可以遵循相同的过程来计算关于X的函数的方差近似值。该计算的重要恒等式是$Var(a+b\times X)=b^2\times Var(X)$，其中“a”和“b”都是常数。取f(X)关于E(X)的一阶泰勒展开式的方差可以得到：

$$Var(f(X))=Var(E[f(X)])+(X-E[X])\times f'(E[X])$$

$$=f'(E[X])^2\times Var(-E[X]+X)=f'(E[X])^2\times Var(X)$$

最终的公式与上面给出的方差近似公式相同。这两个近似公式尽管并不是最精确的，却往往对于快速计算关于随机变量的复杂函数的均值和方差很有用。例如，当函数为log(X)时，均值和方差的近似值为：

$$E[\log(X)]=\log(E[X])$$

$$Var(\log[X])=Var(X)/(E[X]^2)$$

该方法也可以用来计算第10章中某些算法的存储空间需求的近似值。

9.3.3　分布不等式

当使用随机数和分布时，有几个不等式可能有用。这些分布可以用来得到估计值的界，第10章会介绍这一点。

首先介绍的两个不等式为随机变量大于a的概率设定一个上界。第一个不等式是马尔科夫不等式（Markov inequality），即$P(X\geq a)\leq E[X]/a$。

用$(X-E[X])^2$作为马尔科夫不等式中的随机变量，就可以得到切

比雪夫不等式（Chebyshev inequality）：$P(|X - E[X]| \geq a) \leq Var(x)/a^2$。也可以写作$P(|X - E[X]| \geq k \times s) \leq 1/k^2$，其中s是随机变量的标准差。

尽管如果知道随机变量的实际分布，我们通常可以得到更精确的界，这两个不等式都不需要假定随机变量服从特定的分布。只需要保证一阶矩和二阶矩是有限的就可以。

如果X是n个独立的随机变量的和，那就可以得到随机变量的切诺夫界（Chernoff Bound）。切诺夫界提供了随机变量的上界和下界：

$P(x \geq a) \leq e^{-ta} M(t)$，$t > 0$

$P(x \leq a) \leq e^{-ta} M(t)$，$t < 0$

M（t）是前面介绍的矩母函数。对于给定的分布，选择能使上下界的概率最小的t。

最后，可以用詹森不等式（Jensen’s inequality）来处理随机变量的函数。该不等式指出，X的函数的期望始终不小于X期望的函数：$E[f(X)] \geq f(E[X])$。

9.4 随机数产生器

在介绍抽样方法和抽样的相关问题之前，需要讨论随机数产生器。许多语言默认的随机数产生器是线性同余随机数产生器（Linear Congruential Generator Random Number Generator，LCRNG）。该产生器的实现非常简单，如下面的代码片段所示，其中m、a和c是常数，它们的值取决于编程语言：

```
public class LCRNG {

  long x = System.currentTimeMillis();
  long a,c,m;
  public LCRNG(long a,long c,long m) {
    this.a = a;this.c = c;this.m = m;
  }

  public long next() {
    x = (a*x + c) % m;
    return x;
  }
}
```

显然，该产生器非常容易实现。由于它只包含了3个操作，因此速度也非常快但对于统计应用来说并不是很好。之所以得名伪随机数产生器（如果知道起始值，就可以重新生成整个随机数序列，这在测试随机算法时很有用），因为基于算法的随机数产生器都会表现出不同周期长度的周期性行为。对于LCRNG来说，它的周期很短，对于随机数的低阶位来说可能非常短。例如，如果m是2的幂，LCRNG交替生成奇数和偶数，这在酒吧里赌博时有用，但没什么别的用处。

为了解决这个问题，多数产生器只使用LCRNG的高阶位。例如Java在

其默认的随机数产生器中使用48位状态，但只返回48位中的32位。这样就可以丢弃某些周期很短的低阶位。除非应用可以使用的内存很受限，例如嵌入式系统，更好的解决方法是使用更好的随机数产生器。其中最流行的一个是梅森旋转（Mersenne Twister）算法。它是R统计编程语言和MATLAB的默认算法。这个算法不适合加密应用，因为如果观察到产生器生成的足够数量的值就可以预测未来值，这跟LCRNG类似。不过，这个数量足够大，因此对多数统计应用来说，该算法已经够好了。

许多语言中都包含了梅森旋转算法的实现，许多库中实现了该算法，例如本章中使用的Colt库。

只需要以常见的方式初始化梅森旋转对象，该对象就能以许多种方式提供随机抽取（也成为随机变量），特别是本章剩下内容频繁使用的integer和double形式。

生成特定分布

基本的随机数产生器从均匀分布中随机抽取。如果需要随机抽取（也称为随机变量），就将均匀随机变量或均匀随机变量的组合转换为相应的分布。本节介绍当没有对应的库可用时，从这些分布中随机抽取的简单方法。如果有合适的库，通常应该优先选择库，因为尽管库中的随机变量抽取方法的实现更复杂，但更快。具体来说，它们往往实现ziggurat算法，该算法是抽取随机变量的最快方法之一。

不管怎样，简单的方法都是基于好的均匀随机数产生器，例如上一节介绍的梅森旋转算法：

```
import cern.jet.random.engine.MersenneTwister;

public class Distribution {
  MersenneTwister rng = new MersenneTwister();
```

指数分布随机数和泊松分布随机数

如果分布的CDF有可逆的封闭解，从该分布中抽取随机数就很简单。由于CDF的值始终处于0~1之间（包含0和1），因此只需要从[0，1]中抽取均匀随机数，然后将其代入CDF的逆形式来计算x的值。

从指数分布中抽取随机值就是这样做的。回想一下本章前面提到的概率密度函数，指数分布的累积密度函数就是$u = 1 - e^{-px}$。

使用一些简单的操作就可以得出x的计算公式，即给定u，有$x = -\log(1 - u)/p$。由于u是均匀随机数，因此可以直接用1 - u替换u，从而得到从指数分布中随机抽取的简单实现：

```
public double nextExponential(double p) {
    return -Math.log(rng.nextDouble())/p;
  }
```

要生成服从泊松分布的随机变量，可以采用与之非常相似的方法。对于泊松分布，CDF包含了求和步骤，因此要根据k的值迭代，直到找到相应的值。实现代码如下所示：

```
public int nextPoisson(double p) {
    int    k = 0;
    double c = 0, u = rng.nextDouble()/Math.exp(-p);
    while(true) {
      double x = c + Math.pow(p, k)
/Arithmetic.factorial(k);
      if(x > u) return k;
      c += x;
      k++;
    }
  }
```

正态分布随机数

如果累积密度函数没有易用的封闭解，有时可以利用均匀抽取的其他变换来生成服从该分布的随机抽取值。对此，一个经典的例子就是从正态分布中抽取随机值的Box-Muller方法。

其变换方式是，首先得到极坐标系中的一点。在[0，2pi]中抽取一个均匀分布变量作为单位圆中均匀分布的角度，该角度就是该点的极角。再从自由度为2的卡方分布中抽取半径的长度（这实际上是p = ~HF时指数分布的特例），这就是该点的极径。然后将该点的极坐标转换为笛卡儿坐标，从而得到两个独立的正态分布随机变量（其中第2个随机变量可以在下一次随机抽取时使用）：

```
double  z2;
boolean ready = false;
double nextZNormal() {
  if(ready) { ready = false;return z2; }
  double theta = nextUniform(0,2*Math.PI);
  double r     = Math.sqrt(nextExponential(0.5));
  z2 = r*Math.sin(theta);ready = true;
  return r*Math.cos(theta);
}
```

这时生成的是服从“标准正态分布”的随机值，均值为0，标准差为1。也可以对其进行平移和缩放来得到任何服从正态分布的随机值：

```
public double nextNormal(double mu,double sig) {
  return mu + sig*nextZNormal();
}
```

gamma分布、卡方分布和beta分布随机数

尽管gamma分布与指数分布紧密相关，但生成服从gamma分布的随

机数却是比较困难的。如果形状参数k是整数，那么可以直接用相应的尺度参数来生成k个服从指数分布的随机值，然后将其加起来，就能得到服从gamma分布的随机值。

当然，这种方法有不少缺点，其中最主要的就是当k不是整数时无法使用。更常用的方式是使用所谓的拒绝抽样（reject sampling）技术。

在拒绝抽样中，选择另一种更易于抽样的分布f（x），该分布将目标分布g（x）封装起来。gamma分布的封装分布的概率密度函数实现如下：

```
protected double gamma_f(double x,double a) {
  double L = Math.sqrt(2*a - 1);
  double y = Math.log(4) - a + (L+a)*Math.log(a)
      + (L-1)*Math.log(x) - Gamma.logGamma(a)

      - 2*Math.log(Math.pow(a, L) + Math.pow(x, L));
  return Math.exp(y);
}
```

可以用从指数分布中抽取随机值的逆CDF方法来生成服从该分布的随机值：

```
protected double nextGammaF(double a) {
  double L = Math.sqrt(2*a - 1);
  double u = rng.nextDouble();
  return a*Math.pow(u/(1-u), 1/L);
}
```

然后以g（x）/f（x）的概率用封装分布的随机值作为目标分布的随机值，如果没有命中概率，就尝试另外一次抽取，直到条件满足为止。

```
protected double nextGamma(double k,double p) {
    while(true) {
      double x = nextGammaF(k);
      if(rng.nextDouble() <= gamma_f(x,k)/Distributions.dgamma(x, k, 1)) {
        return x/p;
      }
    }
  }
```

注意，g（x）/f（x）越接近1，算法的性能就越好。图9-1展示了不同形状参数的gamma分布的概率密度。虚线是具有相同形状参数的拒绝抽样方法的封装分布的概率密度。这里选择的f（x）非常好，如图9-1所示，但从gamma分布的尾端抽样的效率会比较低。人们已经开发了其他方法来提高效率。

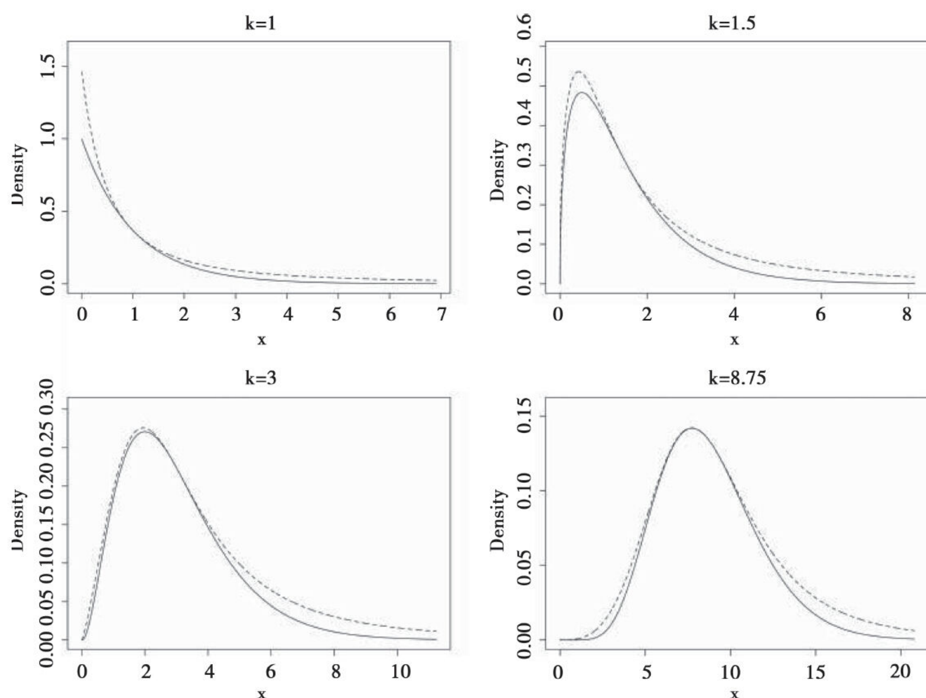


图　9-1

有了从gamma分布中抽样的方法，从卡方分布和beta分布中抽样就很简单了。显然，卡方分布就是gamma分布的特例，因此可以直接生成随机值：

```
public double nextChiSq(double k) { return nextGamma(k/2,0.5); }
```

对服从beta分布的随机变量的抽取，也要利用它与gamma分布的关系，这里是取一个gamma分布之和与两个gamma分布之和的商：

```
public double nextBeta(double a,double b) {
    double x = nextGamma(a,1);
    double y = nextGamma(b,1);
    return x/(x+y);
  }
```

9.5 抽样过程

到目前为止，本章已经介绍了随机变量的概念，并展示了怎样从这些分布中“抽取”随机值。当数据量很大时，也可以通过从数据中抽样来生成随机抽取值。几十年以来，人们开发了许多抽样过程，发表了大量相关论文。这些过程许多都聚焦于在普查全部数据代价太大或太耗时（或者两者兼而有之）的情况下进行调查问卷这样的问题。由于本书主要关注流数据，因此其中的多数过程都超出了本书的讨论范围。

本节介绍了从固定规模的数据中进行抽样的基础知识，基于这些知识你可以理解怎样从流数据集中抽样。然后给出了对基础流过程的一些修改，使其能在一些更有趣的流分析场景中运行。

9.5.1 从固定数据集中抽样

不出意外，最简单的抽样称为简单随机抽样（simple random sampling）。该抽样过程的目标是，从有N个元素的数据集中抽样n个元素，并保证任何给定的元素都有相同的抽样机会。如果所有元素都存放在内存中，并且允许给定元素在抽样集中出现不止一次，抽样的实现就很简单：

```
public class SimpleRandomSample {
  static MersenneTwister rng = new MersenneTwister();
  public static <E> E[] withReplacement(E[] in,int n) {
    Object[]      sample = new Object[n];
    for(int i=0;i<n;i++)
      sample[i] = in[(int)Math.floor(n*rng.nextInt())];
    return (E[])sample;
  }
}
```

如果整个数据集都可以放入内存，但元素要进行无放回（without replacement）抽样，抽样算法就略有不同。该算法的基础是Fisher-Yates混洗法，这是20世纪30年代末提出的纸笔问答方法。这是个实现起来很简单的混洗方法：

```
public static <E> void permute(E[] array) {
    for(int i=0;i<array.length;i++) {
      int j = i + (int)Math.floor((array.length - i)*rng.nextDouble());
      E temp = array[i];array[i] = array[j];array[j] = temp;
    }
  }
```

该方法会以相同的概率生成数组的全部n！个排列。在上述算法中，一个值被交换的次数不会超过一次，因此可以对前n个元素运行混洗算法并返回前n个元素作为抽样集，从而实现无放回抽样：

```
public static <E> E[] withoutReplacement(E[] array,int n) {
  for(int i=0;i<n;i++) {
    int j = i
+ (int)Math.floor((array.length - i)*rng.nextDouble());
    E temp = array[i];array[i] = array[j];array[j] = temp;
  }
  return Arrays.copyOf(array, n);
}
```

9.5.2　从流数据中抽样

当数据规模已知且数据可以全部放入内存中的数组中时，Fisher-

Yates混洗法很好用。如果数组无法全部放入内存，就要开发其他技术来支持单遍的数据抽样。这些方法构成了流数据抽样的基础，流数据的整个数据集规模介于“非常大”和“无限”之间。

水库抽样算法

水库抽样（reservoir algorithm）指的是一类流数据抽样算法。它们在概念上与上一节的Fisher-Yates混洗法紧密相关。通过早期的Fisher-Yates混洗算法可以看出两者的关系：

```
public static <E> void durstenfeldPermute(E[] array) {
  for(int i=array.length-1;i>0;i--) {
    int j = (int)Math.floor(i*rng.nextDouble());
    E temp = array[i];array[i] = array[j];array[j] = temp;
  }
}
```

在这个版本的算法中，索引是从后向前混洗的。这意味着数组中较远的元素被换到数组的前面。就是这个概念启发了基本的水库抽样。

在水库抽样算法中，首先分配一个由元素组成的水库抽样数组。该数组的作用类似于无放回抽样中的前n个数组元素。

```
public class Reservoir<E> implements List<E> {

  MersenneTwister rng = new MersenneTwister();
  Object[] reservoir;
int       N;

public Reservoir(int n) {
  super();
  reservoir = new Object[n];
  N          = 0;
}
```

前reservoir.length个元素可以直接加入数组。之后，生成N~0之间的

一个随机位置。如果该位置落入数组中，就用新元素替换当前位于该位置的元素；否则，就不将新元素加入。

```
public boolean add(E arg0) {
    if(N < reservoir.length) {
      reservoir[N] = (Object)arg0;
    } else {
      long k = (long)Math.floor(N*rng.nextDouble());
      if(k < (long)reservoir.length) reservoir[(int)k] = (Object)arg0;

  }
  N++;
  return true;
}
```

水库抽样从后向前混洗整个数组，这实际上与Durstenfeld提出的改进的Fisher-Yates算法的做法相同。这里没有给出证明，但这确保了在分析过程中的每个时刻，流中的每个元素都有相同的抽样概率。

9.5.3 有偏流抽样

基本的水库抽样也称为R算法（Knuth的著作《The Art of Computer Programming》给出的名字），它确保了从流开始到结束都均匀抽样。这主要归因于它最初的应用，该应用从“流”中抽样，只允许对有限数据集进行单遍处理。这使得该算法非常适合在用Map-Reduce处理有限数据集的批处理环境下抽样。

对于本书所用的流概念来说，该算法就没有那么大的吸引力了。本书更关注抽样随着时间不断变化的动态性。实际上，随着抽样的老化，最近的数据点会被包含到抽样中的可能性会越来越小（随着N变大，[0，N]中的随机数小于n的概率会变小）。取而代之的是，如果与较旧的事件相

比，算法可以偏向于将较新的事件包含在抽样中，那么算法更可取。

9.5.3.1 滑动窗口水库抽样

将时间动态性引入水库抽样的一个简单方法是维护一个抽样滑动窗口。在该实现中，用链表维护了多个水库抽样“桶”（buckets）：

```
public class SlidingWindow<E> {
  int n,k,max;
  int last = Integer.MAX_VALUE;

  public SlidingWindow(int n,int k,int max) {
    this.n = n;this.k = k;this.max = max;
  }
```

该实现有3个参数：n是每个水库抽样的大小；k是各桶起点之间的间隔；max是在每个桶过期之前考虑的最大元素数量。如果将所有这些参数都置为相同的值，那么与其说是抽样，还不如说是将数据分桶存放的方法。

每次插入时，有几个不同的步骤。如果已经加入超过k个元素，那就生成一个新桶。然后，将过期桶从活跃列表中删除，将其放入completed队列，由另外的线程作进一步处理。最后，才将元素加入每个活跃桶中。

```
public void add(E object) {
  if(last >= this.k) {
    last = 0;
    live.add(new Reservoir<E>(n));
  }
  while(live.peek() != null && live.peek().total() >= max)
    completed.add(live.poll());
  for(Reservoir<E> e : live) e.add(object);
  last++;
}
```

使用该方法，每个抽样都只包含特定时间范围内的抽样。然后就可以

将这些抽样单独处理，以生成随时间变化的估计值。

9.5.3.2 指数有偏抽样

尽管滑动窗口易于实现，但也会占用大量内存。当抽样和滑动窗口有大量重叠时尤为如此。此外，它还会引入计算量与活跃窗口数量呈线性关系的操作。

另外一种方法是只维护一个抽样，根据抽样中的元素与新元素年龄的差别，通过用新元素替换原有元素，来保证抽样中倾向于包含较新的元素。本节介绍Charu Aggarwal提出的方法（“On Biased Reservoir Sampling in the Presence of Stream Evolution，”VLDB Conference，2006）。

该方法首先定义偏置率p，其函数形式与指数分布相同。该算法用偏置率的倒数来定义水库抽样的大小：

```
public class BiasedReservoir<E> {
  MersenneTwister rng = new MersenneTwister();
  ArrayList<E> reservoir = new ArrayList<E>();
  double rate;
  double size,N;

  public BiasedReservoir(double rate) {
    this.rate = rate;
    size      = Math.ceil(1.0/rate);
  }
```

注意，这里的水库抽样实际上并不是最初的数组形式。这是因为可以允许数组中的元素数量随着时间发生变化。为了实现add（）方法，首先以一个概率删除一个元素，该概率等于水库抽样中的元素数量除上水库抽样的大小。如果元素数量等于水库抽样大小，那么元素始终是被删除的。

要插入的元素就会被加入到数组中：

```
public void add(E obj) {
  if(rng.nextDouble() < ((double)reservoir.size())/size) {
    reservoir.remove(rng.nextInt() % reservoir.size());
  }
  reservoir.add(obj);
}
```

Aggarwal证明了该简单算法以等于水库抽样“大小”的速率生成指数有偏抽样。该方法的主要缺点是需要一定的时间才能完全填满水库抽样。在多数流应用中，计算是耗时的，因此流的初始化通常不成问题。

9.6 总结

本章对统计学和概率论领域进行了宽泛的介绍。尽管这些知识并不是专门针对流数据分析的，但它们是分析来自总体的抽样的关键。在某种程度上，任何数据流都是从所有可能的数据流构成的集合中抽取的样本，但即使将数据流视为数据集，它也太大，无法直接操作。

由于数据流太大，很难直接使用，本章还介绍了一些简单的抽样方法。这些样本可以来自非常大但有限的数据集，也可以来自真正的无限数据流。下一章采取了另一种策略，主要关注以“有损”方式对数据流进行汇总的方法。在统计学中，这称为“降维 ”（dimension reduction）。然后，最后一章不再简单罗列数据，而是应用本章和第10章的技术实际地分析数据。

第10章　使用略图近似流数据

在数据分析任务中，经常要询问由不同元素组成的集合的相关问题，这样的问题可能是流式的，也可能不是。常见的问题通常是“是否存在这个元素？（集合成员查询）”、“总共有多少个不同的元素？”（基数估计）或“这个元素出现了多少次？”（频率）。

当集合中不同元素的数量较小时（基数小），即使在流系统中也可以直接计算。前面几章中介绍的某些存储机制，如Redis，甚至有专门的数据结构，用来维护实时更新的集合和直方图。

但是，当集合的基数变得很大时（也就是说，有许多不同的元素），直接维护集合就很成问题。维护这些数据结构需要O（logn）的更新时间和O（n）的存储空间，因此在流环境下不可行，并且（起码）硬件成本也很高。

为了解决这个问题，人们开发了统称为略图（sketch）的一类算法，以计算这些问题的近似结果。略图算法之所以可取，是因为它具有三个特点。首先，略图的数据更新时间是恒定的，这使得它们在流环境下很容易维护。其次，略图需要的存储空间通常与数据量无关。最后，对该数据结构的查询最差情况下也需要线性时间。略图的缺点是，为了满足上述特点，就必须向结果中引入误差。在这一点上，略图算法与第9章讨论的抽样方法类似。

本章讨论了四种略图算法，每种算法的关注点和性能特征都各不相

同。Bloom Filter是通用的集合表示方法，可以用来回答集合成员和基数的问题。它也是这些算法中最古老的，许多领域都要用到它的变体。两个不同值略图（Distinct Value Sketch），即Min-Count和HyperLogLog算法，使用不同的方法对集合的大小（即基数）进行近似。最后，Count-Min略图对由集合中元素的频率组成的多集（multi-set）进行近似。如果多集中的元素是有序的，多集就可以近似表示直方图。

本章首先概述所有略图算法所共有的概念背景。具体来说，这些算法都频繁使用哈希函数。哈希函数在计算机科学的许多方面都发挥着重要作用，这里，略图应用用到了它们的统计特性。本章还回顾了关于数学集合的一些内容。这是因为本章的多数算法都涉及集合运算。这些算法的性能的提高有赖于集合的一些重要性质。

10.1　寄存器和哈希函数

所有的略图都使用相同的基本构件来实现自己特殊的算法，这两个基本构件是寄存器（register）和哈希函数。本节讨论这两个基本构件的性质，其中重点是哈希函数。

10.1.1　寄存器

寄存器 是一个很简单的概念，它们就是用来存储略图数据的计数器数组。在多数略图应用中，相比直接表示数据集合所用的数组，寄存器需要的存储空间要小得多。它们只需要恒定的存储空间，并且该空间的大小与输入元素的数量无关。寄存器存储的数值范围通常比原始输入要小，这样就可以用比原始输入少的位数来表示寄存器。例如，基本的Bloom Filter值只能将寄存器置为1或0，因此可以使用位数组而不是字节数组，这样存储空间需求就降为原始输入的1/8。

10.1.2　哈希函数

哈希函数只有一个输入，不论输入的长度如何，它都返回m个输出中的一个输出，该输出的值属于一个有限空间。例如，哈希函数的输入可以是一个理论上无限长的字符串，它的输出可以是一个32位整数，因此有 2^{32} 个可能的输出。

哈希函数在计算领域应用广泛，数十年来人们研究和开发了各种各样

的哈希函数。所有的哈希函数都是确定性的：相同的输入总会产生相同的输出。哈希函数也基本上是均匀分布的，这样随机输入产生的输出服从均匀分布。

对于略图应用来说，对任何哈希函数，速度都是最主要的要求，如果是在流环境下使用略图，那就更是如此。这就基本上将MD5和SHA等加密哈希函数排除在外。这是因为，尽管这些哈希函数的随机性非常好，但它们被故意设计为计算起来特别慢。尽管如此，在现实的略图实现中，却往往会使用加密哈希函数，因为它们被内置在许多编程语言的公共库中。

大多数略图实现会使用各种快速哈希函数，而不是使用加密哈希，这些快速哈希函数更能满足性能要求。它们包括FNV哈希、Jenkins哈希和Murmur哈希。Fowler、Noll和Vo（FNV）哈希函数都开发于1991年，它们都有多个变体，支持32位到1024位输出空间。FNV是实现最简单的哈希函数之一，它依靠从表中选择的两个常数来确定生成结果的位数。例如，64位Java哈希函数的实现如下所示：

```
static long fnv64(byte[] input) {
  long output = 0xcbf29ce484222325L;
  for(var i=0;i<input.length;i++) {
    output ^= (long)input[i];
    output *= 0x100000001b3L;
  }
  return output;
}
```

FNV哈希对于非常小的输入很理想，在Intel处理器上运行速度很快，但在现实应用中，其他两个哈希函数（Jenkins和Murmur）往往具有更好的性能。Murmur哈希似乎在现实应用中效果不错，它有32、64和128位的变体。这里没有给出Murmur哈希的代码，因为这涉及很大的常数数

组，重现起来没有多少意义。但在本书中提供的代码中，我们使用了Colt库提供的Murmur哈希的Java实现。用其他语言编写的Murmur哈希代码也有很多，这些代码的许可方式千差万别。

10.1.2.1 独立哈希

许多算法需要若干独立的或成对独立（pairwise independent）的哈希函数。在计算机科学中，如何生成具有这类性质的哈希函数已经得到了深入研究，研究结果表明这是很难的。因此，多数实际的哈希函数实现转而依靠更容易生成的全域哈希函数（universal hash function）。全域哈希函数就是简单地从一个族中随机选取哈希函数。

全域哈希函数的实现有两种方法。第一种方法最显而易见，即哈希函数以一个初始的种子值作为输入，这种方式很像随机数产生器（本章中使用的哈希函数在实现上与许多随机数产生器都很相似）。要生成k个哈希函数，就从初始值的取值空间（通常是与哈希函数的最大输出值相等的整数）中均匀地选取K个种子。

另一种方法只需要1个种子来初始化哈希函数。用该种子来生成第一个哈希值，然后用生成的哈希值作为初始值，生成下一个哈希函数，依此类推，直到生成第k个哈希值。例如，可以修改前面提到的FNV实现，让它以同样的输入生成任意数量的哈希值：

```
public class FNV {
  public static final long INIT  = 0xcbf29ce484222325L;
  public static final long PRIME = 0x100000001b3L;

  long   hash;
  byte[] input;
  public void set(byte[] input) {
    hash = INIT;
    this.input = input;
  }

  public long next() {
    for(int i=0;i<input.length;i++) {
      hash ^= input[i];
      hash *= PRIME;
    }
    return hash;
  }
}
```

10.1.2.2 双哈希

为了进一步提高哈希函数的计算速度，Kirsch和Mitzenmacher于2007年提出，使用称为双哈希（double hashing）的技巧就可以生成任意数量的哈希函数。双哈希这个名字的意思是，只需要用两轮基本的哈希函数。它们还证明了，该方法与生成k个独立的哈希函数相比，性能上的差别很小，因此适于在现实环境下使用。

双哈希常用来解决哈希表中的冲突问题，它将哈希函数定义为g（x，i）=h1（x）+i*h2（x）+i^2。在这里，h1和h2是先前的两轮初始哈希函数。这里i是0到k - 1之间的整数，k是想要的哈希函数的数量。计算前两个哈希值之后，再计算最后的哈希值就代价就不大了，只需要两次乘法，而不用进行多轮昂贵的哈希函数计算。

HASH冲突和生日悖论

有一个著名的概率方面的小把戏，即所谓的生日悖论。当有超过23个人在一起时，至少两个人的生日是同一天的可能性超过50%。如果有60个人，这个可能性就超过99%。

可以将生日看作简单哈希函数，该函数将每个人映射到1到365之间的某个数字。这样，生日相同就可以看作哈希冲突。之所以举这个例子，是因为了解该悖论背后的数学知识，有助于理解后面几节的许多计算问题。除此之外，这也是聚会中很讨彩的小把戏。

首先，假设生日在一年中是均匀分布的，也即哈希函数的值是均匀分布的。现实中，生日在一年中并非是精确的均匀分布，并且闰年也是个问题。这里不深入讨论这个问题。

要计算至少两个人的生日是同一天的概率，最简单的方式是计算房间中没有任何人的生日是相同的概率。这样，第一个人可以任意选择，这样她的生日就可以是任意一天A。第二个人不能选择与第一个人相同的生日，因此只能从剩下的364天中选择一天B作为生日。第三个人选择生日C，依此类推，直到第n个人。概率的计算如下：

P（无人生日相同）

=（365/365）*（364/365）*…*（（365 - n - 1）/365）

该公式可以重写为：

P（无人生日相同）= 365！/365^n （365 - n）！

其中！表示阶乘函数，即从1到n所有整数的乘积。因此，5！为

5×4×3×2×1。

最后，至少两个人的生日是同一天的概率就是1 - P（无人生日相同）：

1 - 365！/365^n（365 - n）！

上述最终公式包含了计算代价很高的阶乘函数。为了简化计算，往往要进行近似。可以将（365 - n - 1）/365改写为（1 - （n - 1）/365），并且我们知道，当x远小于1时e^x近似等于1 + x（通过e^x的一阶泰勒展开得到）。将这两者结合起来，$e^{-(n-1/365)}$就近似等于（1 - （n - 1）/365）。因此可以将概率重写为：

$e^{-(1+2\ldots+(n-1)/365)}$

由于从1到n - 1的整数的和为n×（n - 1）/2，上述公式可以进一步简化为：

$e^{-n*(n-1)/2*365}$

如果n足够大，n和n - 1之间的差会非常小，这样就可以将n*（n - 1）替换为近似值n^2。将该公式的值设为0.5，可以求得n等于23，也就说，如果房间里有23个人，那么其中至少两个人生日相同的概率就会超过50%。

10.2 集合

略图算法常常要与集合打交道，特别是经常要近似计算集合的规模，这种计算称为基数估计。本节回顾了“朴素集合论”中的一些知识，在使用或分析后面几节讨论的算法时，这些知识会派上用场。

集合（set）是由不同的元素组成的集体。集合可以是空的，称为空集（empty set）或零集（null set）。通常用A或B这样的大写字母来表示集合。用空符号（$\varnothing$）表示空集合。

注意　本书用到了一些基础的数学术语和公式。如果你对这些数学知识感到生疏，对于这些概念的理解有些困难，可以参考Sheldon Ross的《A First Course in Probability》一书。

有三种方式可以将集合结合在一起。集合的并（union）$A \cup B$）包含的是A和B的全部元素。集合的交（intersection，$A \cap B$）包含的是A和B共同拥有的元素。集合的补（complement $A \setminus B$）包含的是属于集合A但不属于集合B的元素。

在Java中，addAll（）、retainAll（）和removeAll（）分别对应了基本的并、交和补方法。可以使用这些基本方法，对任意数量的集合实现并运算和交运算：

```
public static Set<E> union(Set<E>...sets) {
  Set<E> union = new Set<E>();
  for(Set<E> s : sets) union.addAll(s);
  return union;
}

public static Set<E> intersection(Set<E>...sets) {
  Set<E> intersection = new Set<E>();
  intersection.addAll(sets[0]);
  for(int i=1;i<sets.length;i++)
    intersection.retainAll(sets[i]);
  return intersection;
}
```

集合中元素的数量称为集合的基数（cardinality）。将集合用“|”符号包围起来就表示该集合的基数。例如，集合A的基数记为|A|。在Java中，用size（）方法返回集合基数。

计算集合的并集或交集的基数更有趣一些。在本章后面给出的多数算法中，计算两个或多个集合的并集是很简单的，而计算交集则有点难（如果存在交集的话）。直观地说，原因在于后面所讲的算法都支持向集合中加入新元素。这等价于计算原集合与新集合的并集，而新集合只包含要加入的新元素。但是，多数这些算法不支持将元素从集合中删除，而这正是计算交集所必需的操作。此外，许多算法可以不用显式地计算并集的表示，就得到并集的基数，从而使得我们可以用很小的计算代价获得并集的基数。

要计算交集的基数，需要利用容斥原理（inclusion-exclusion principle）。该原理用最简单的形式，给出了两个集合并集的基数与交集的基数之间的关系：

$$|A \cup B| = |A| + |B| - |A \cap B|$$

这个关系本质上说的是，如果将A的基数和B的基数相加，那么交集中的元素就会被计数两次。要纠正这个问题，那就得减去交集的基数。改写这个公式，就可以得到计算交集基数的公式：

$$|A \cap B| = |A| + |B| - |A \cup B|$$

用相同的原理可以得到三个集合交集基数的计算公式：

$$|A \cap B \cap C| = |A \cup B \cup C| - |A| - |B| - |C| + |A \cap B| + |A \cap C| + |B \cap C|$$

用第一个公式来替代其中的交集，去掉其他的交集，就得到最后的公式：

$$|A \cap B \cap C| = |A \cup B \cup C| + |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C|$$

还可以进一步推广，得到任意数量集合的并集基数的表达式：将所有集合的基数相加，减去每两个集合交集的基数，加上每三个集合的交集的基数，减去每四个集合交集的基数，依此类推。

尽管通过这种方法，可以近似得到任意个集合之间交集的基数，但它有两个缺点：

·需要的操作总数很快就会变得很大。

·本章介绍的Bloom Filter和Distinct Value略图算法都有自己的误差，并且误差会随着每次加减操作而累积。即使估计两个集合交集基数的误差，也会是估计任何一个集合或这两个集合并集误差的三倍。

10.3 Bloom Filter

Bloom Filter是一种数据结构，它被许多应用用来存储集合成员信息。该数据结构很简洁，并与集合中存储的元素数量无关。它也因此付出代价，可能会错误地将本不属于集合的某个元素报告为属于该集合，即假阳性（false positive）。但Bloom Filter不存在假阴性（false negative）。它的数据结构的大小决定了假阳性率。

10.3.1 算法

Bloom Filter用m个单位寄存器组成的数组，以及K个独立的哈希函数近似地表示一个集合。在Java中，可以用BitSet和k个Murmur哈希函数来实现java.util.Set＜E＞接口，其中，这k个哈希函数每个都有独立的随机值作为种子：

```
public class BloomSet<E extends Serializable> implements Set<E> {

  BitSet                 bits;
  int                    m;
  SerializableHasher[] hashes;
  ObjectOutputStream[] outputs;

  protected void initialize(int m,int[] seeds) throws IOException {
    this.m = m;
    bits   = new BitSet(m);
    hashes = new SerializableHasher[seeds.length];
    outputs= new ObjectOutputStream[seeds.length];
    for(int i=0;i<seeds.length;i++) {
      hashes[i]  = (new SerializableHasher()).seed(seeds[i]);
    outputs[i] = new ObjectOutputStream(hashes[i]);
  }
  bits.clear();
}

public BloomSet(int m,int[] seeds) throws IOException {
  initialize(m,seeds);
}
```

SerializableHasher和ObjectOutputStream数组使用Java序列化机制实现32位Murmur哈希，这样，任何实现Serializable的对象都可以使用该类。

在实际工作中，实现专门的集合来存储类似于String或Long类型的值也很常见。这些数据类型在流分析中经常出现，因此使用专门的类可以大幅提高性能。实际上，本书提供的示例代码中使用的Murmur哈希实现，就有针对这些使用场景优化的专门版哈希函数。

向集合中加入新元素是很容易的。首先，使用各哈希函数来生成k个不同的哈希值，再用取模运算符将这些哈希值映射到m个寄存器组成的数组中，最后将寄存器数组中所映射位置上的位置为1。该方法的Java实现如下所示：

```
public boolean add(E arg0) {
  if(!contains(arg0)) {

    for(int i=0;i<outputs.length;i++) {
      hashes[i].reset();
      try {
        outputs[i].writeObject(arg0);
        outputs[i].flush();
        bits.set(hashes[i].hash() % m);
      } catch(IOException e) { }
    }
    return true;
  }
  return false;
}
```

上述代码实现中使用contains函数对元素进行检查，返回的结果是该元素是否应被视为新元素。被视为新元素的条件是，k个哈希函数计算出来的所有的寄存器都是1。下面是contains函数的实现代码：

```
public boolean contains(Object arg0) {
  for(int i=0;i<outputs.length;i++) {
    try {
      hashes[i].reset();
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      if(!bits.get(hashes[i].hash() % m)) return false;
    } catch(IOException e) { }
  }
  return true;
}
```

看过上述代码，假阳性的存在就很容易理解了。随着越来越多的元素加入集合，下面这种情况出现的可能性会越来越高：要加入集合的新元素所映射到的寄存器可能都已经被其他元素映射过。一旦出现这种情况，即使还没有将该新元素加入集合，contains（ ）实现也会返回true，这样就出现了假阳性。

在在线广告中过滤重复事件

在线广告中一个常见的做法是，为一则广告的每次浏览赋予一个唯一的标识符。这个唯一标识符用来识别只对广告浏览或点击了一次，却向系统注册了多次的情况。产生这种情况的原因有很多种，可能只是用户习惯性地双击链接等常规情形，也可能是恶意“僵尸程序”为了获得非法收益，将广告投放在欺诈网站上。

在确认页面浏览（即印象（impression））和点击的有效性时，广告行业通常比较保守。如果过于谨慎而丢弃小部分有效事件，却可以清除掉更多无效事件的话，这种做法通常是可取的。另外，广告界也一直在转向所谓的“程序化购买”，即像现实世界的商品交易那样，在互联网上进行广告交易。这种交易是通过竞价机制来进行的，每次竞价的时间通常小于100毫秒。

由于有大量唯一的标识符，并且需要对印象和点击事件维护近乎实时的计数，以支持最优竞价，对这类重复事件进行过滤就成为Bloom Filter这类数据结构的完美应用。

早期的方法通常要维护两个Bloom Filter，一个针对页面浏览（印象事件），另一个针对页面点击。但是，页面浏览几乎占到了所有事件的100%，而页面点击可能只有1%到5%。为了核算的需要，过滤印象事件仍然是必要的，不过，可以通过离线的方式进行。另一方面，页面点击事件会被用于确定竞价，因此可以维护一个Bloom Filter来过滤重复点击事件，从而避免因为这种重复事件，导致广告投放的质量被人为地提升。

10.3.2　Bloom Filter大小的选择

显然，对于上一节末尾提到假阳性场景，其发生的可能性是三个参数的函数：要插入集合中的元素的数量（n）、位数组本身的大小（m），以及用来设置位的哈希函数的数量（k）。后两个参数由算法的用户选择，第一个参数则通常是估计得来的。本节讨论后两个参数的选择问题，以及它们对假阳性率的影响。

根据前面的介绍，要正确地识别出某个元素不在集合中，在k个哈希函数确定的位中，至少有1位必须是0。如果哈希函数是相互独立且均匀分布的，在向过滤器中插入n个元素之后，其中1位仍然是0的概率p应为$p = (1 - 1/m)^{kn}$。该等式可以近似为$p = e^{-kn/m}$。假阳性的概率近似等于$(1 - p)^{k}$，即要检查的所有k个寄存器都已经被置为1的概率。这样，假阳性概率的最终结果就是$(1 - e^{-kn/m})^{k}$。

我们的目的是使p尽可能小，此时$k = (m/n) \times \ln 2$。因此，使用的哈希函数的数量取决于过滤器的大小，以及进入Bloom Filter的元素数量的估计值。将其代入前面式子，就可以得到：

$$P = (1 - e^{-\ln 2/n})^{(m/n)*\ln 2}$$

求出m：

$$m = -n \times (\ln p) / (\ln 2)^{2}$$

将n设为1，那么存储1个值需要的位数近似为$2.08*\ln(1/p)$或$1.44*\log 2(1/p)$。根据这个近似公式，如果假阳性率为5%，对每个要进入集合中的元素，需要6.22位的存储。将假阳性率减半，只会使每个元素需要的存储增加到7.7位。如果假阳性率为1%，存储每个元素也只需要

9.6位。

当然，分配给寄存器数组的位数量必须是整数。因此，常见的方式不是指定错误率，而是指定每元素对应的位数量c，然后据此计算位数组的大小，以及哈希函数的数量。给定元素的目标数量，整个寄存器数组的大小就是$m = cn$。要使用的哈希函数的数量就是$k = c \times \ln 2$。可以很快算出预期的假阳性率近似为$p = 0.6185^c$。举例来说，如果每个元素对应8位，那么$k = 6$，假阳性率近似为2.14%。

看起来这似乎没有太大进步，但在现实应用中，这种改进是巨大的。例如，最长的域名为63个字符，实际应用中最短域名为3个字符。通过大量观察得到证据显示，域名的平均长度约为10个字符。这样，如果跟踪一百万个域名，应用需要约10MB的存储空间。而如果用Bloom Filter跟踪同样这么多域名，假设错误率小于2.5%，那么只需要不到1MB的存储空间。

10.3.3　并集和交集

Bloom Filter有一个很好的性质，如果两个Bloom Filter是用相同的哈希函数构造的，那么可以分别使用按位或（OR）或按位与（AND）运算符，来得到它们的并集和交集。

在计算并集的情况下，按位或操作构造的Bloom Filter等价于以相同的假阳性率，直接从两原始集合的并集来构造的Bloom Filter。这并不令人意外，因为第j次操作实质上是取包含1个元素的Bloom Filter与包含前j - 1个插入元素的Bloom Filter的并集。这个操作很方便，因为它使得我们

可以先以分布式的方式构造过滤器，再通过交换这些比原来小得多的过滤器，将其合并起来，从而构造出整个过滤器。

两个Bloom Filter的交集要复杂一些。此时，这个交集的假阳性率不会超过原来两个Bloom Filter的假阳性率，但可能要高于从原始集合的交集直接构造出来的Bloom Filter的假阳性率。

10.3.4　基数估计

对于维护集合基数的估计值，有许多有效的方法。Bloom Filter也是其中之一。

基本的技巧是，利用前面提到的Bloom Filter的特定位被置为1的概率。当选择了过滤器大小以及哈希函数数量的最优值时，这个概率为$p = 1 - e^{-kn/m}$。

假设各个位被置为1的概率是相互独立的，那么就可以将寄存器数组视为具有m个独立且相同分布的伯努利试验。这等同于计算m次抛硬币有多少次正面朝上。

插入n个元素之后，可以估计有1位为1的概率，该概率等于位为1的数量x除上位的总数m。将x/m代入上述等式，可以求得n：

$$N = -m \times \log(1 - x/m)/k$$

如果不使用前面分析的近似形式，可以获得更好的估计值。这时n的表达式要复杂一些：

$N = \log(1 - x/m) / (k \times \log(1 - 1/m))$

但是，在现实情况下，$-m/k$与$1/k \times \log(1 - 1/m)$之间的差别往往非常小。

Bloom Filter的Set类的size（）方法的原始实现很简单，就是调用BitSet原生的cardinality（）方法：

```
public int size() {
  int X = bits.cardinality();
  return (int) (Math.log(1 - (double)X/
  (double)bits.length())/
  (Math.log(1.0 - 1.0/
  (double)bits.length())*(double)hashes.length));
}
```

Cardinality（）的实现通常使用并行计数的技巧来提高性能，但仍然需要O（m）的运行时间。如果该方法要被调用许多次，可以将X的计算集成到add（）方法中，从而获取更好的性能，如下所示：

```
public boolean add(E arg0) {
  if(!contains(arg0)) {
    for(int i=0;i<outputs.length;i++) {
      hashes[i].reset();
      try {
        outputs[i].writeObject(arg0);
        outputs[i].flush();
        int h = hashes[i].hash() % m;
        X += bits.get(h) ? 0 : 1;
        bits.set(h);
      } catch(IOException e) { }
    }
    return true;
  }
  return false;
}
```

当过滤器重置时，也要修改reset（）方法，将X置为0。

对欺诈网站的实时识别

欺诈网站生成流量的一种方式就是，利用所谓的“僵尸网络”来为网站生成实质上是伪造的大量流量。这类流量其实很难检测，因为僵尸网络中的多数傀儡机是互联网中的合法计算机，只是不幸感染了恶意软件而已。实际上，在平时，这些设备的合法用户都是以正常而非欺诈的方式运行这些傀儡机的。只有当僵尸网络被激活时，我们才有可能将这些流量确定为欺诈流量。

为了解决这个问题，你需要在特定网站的行为表现与异常行为，即被僵尸网络控制来产生收益时的行为一致时，实时地将这些网站列入黑名单。

为此，首先应获取以前认定的欺诈网站所访问的IP地址模式，来对僵尸网络流量生成一个基础的配置文件。这可以通过人工调查来获取，也可以寻找业界维护的此类数据。

接下来，需要一个距离度量，这样你才能比较两个IP地址配置文件。其中一个这样的度量是Jaccard指数，该指数的定义是，所比较的两个集合交集的大小除以这两个集合并集的大小。

如果用Bloom Filter来维护网站的配置文件，用另一个Bloom Filter来表示僵尸网络的配置文件，那么就可以将前面推导的等式应用到这两个集合的并集，从而很容易地估计出这两个过滤器并集的基数。实际上，你只需要计算被置为1的位的数量，因此并不用真的计算并集的Bloom Filter，只需要统计被置为1或0的位的数量即可。

为了计算交集的基数，你可以利用集合基数的性质：$|A \cup B| + |A \cap B|$

= |A| + |B|。由此可以得到交集的基数为|A| + |B| - |A∪B|。现在就可以估计任意网站的流量模式与僵尸网络配置文件的Jaccard相似度了。你可以直接将过于相似的网站列入黑名单，或加上标记以做进一步调查。

10.3.5　有趣的变体

Bloom Filter已经出现了很长时间，人们在最初的算法上发展出了有许多变体。本节涵盖了两类比较有趣的变体。具体来说，这两种变体解决的是流环境下容易出现的问题。在此种情况下，随着时间的推移，基本的Bloom Filter很可能日趋饱和，从而难以发挥应有的作用。

10.3.5.1　Counting Bloom Filter

其中一个比较有用的变体是Counting Bloom Filter。引入该变体是为了解决基本的Bloom Filter无法删除元素的问题。

在这个变体中，算法的哈希部分与基本算法相同。选择寄存器数量和哈希函数数量的条件也是相同的。

所不同的是，该变体中的寄存器不再是1位，而是计数器。分配给每个计数器的位数非常小。有证据表明，对多数应用来说，4位就足够了。

如果要加入元素，计数器就直接累加，而不是将位置为1。要删除元素，只要 计数器没有达到最大值，k个哈希函数的每个计数器都递减。如果达到最大值，计数器就被视为溢出，会永久保持其最大值。

在下面的例子中，我们慷慨地为每个计数器分配8位，只需要简单修

改add（E arg0）方法，使得计数器只要没有溢出就一直递增：

```
public boolean add(E arg0) {
  boolean added = false;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int j = hashes[i].hash() % m;
      if(counters[j] < Byte.MAX_VALUE)
        counters[j]++;
      if(counters[j] == 1) added = true;

    } catch(IOException e) { }
  }
  return added;
}
```

Remove（E argo）方法也很容易实现，也需要考虑计数器溢出的情况：

```
public boolean remove(E arg0) {
  boolean removed = false;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int j = hashes[i].hash() % m;
      if(counters[j] > 0 && counters[j] < Byte.MAX_VALUE)
        counters[j]--;
      if(counters[j] == 0) removed = true;
    } catch(IOException e) { }
  }
  return removed;
}
```

集合成员测试的实现也就很简单了，只需要确保所有的计数器都是正数即可，这与基本Bloom Filter的实现相同：

```
public boolean contains(Object arg0) {
  for(int i=0;i<outputs.length;i++) {
    try {
      hashes[i].reset();
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      if(counters[hashes[i].hash() % m] == 0) return false;
    } catch(IOException e) { }
  }
  return true;
}
```

最后，如果要计算基数，只需要检查计算器是否是正数：

```
public int size() {
  double X = 0.0;
  for(int x : counters) if(x > 0) X += 1.0;
  return (int) (Math.log(1 - (double)X
    /(double)counters.length)

    /(Math.log(1.0 - 1.0
    /(double)counters.length)*(double)hashes.length));
}
```

10.3.5.2 Stable Bloom Filter

Stable Bloom Filter解决的是，在时间序列的场景下从Bloom Filter中删除元素的问题。前面我们曾经提到过，如果允许Bloom Filter直接从数据流中获得元素，Bloom Filter可能会直接饱和，从而对任何集合成员查询都会返回肯定的结果，对集合基数查询则会始终返回最大值。

为了解决这个问题，Stable Bloom Filter向过滤器本身引入老化（aging）的概念。该过滤器的基本结构使用与Counting Bloom Filter相同的计数器。为了简化起见，Stable Bloom Filter的实现扩展了Counting Bloom Filter，它增加了两个额外的参数d和C。前者定义了每次增加计数前要进行递减的计数器的数量，后者定义了每个计数器的最大值。这样，通过确保没有递增的值最终会返回0，就可以使过滤器“老化”。该过滤器

的实现代码如下：

```
public class StableBloomSet<E extends Serializable>
  extends CountingBloomSet<E> {

  int  d = 0;
  byte C = Byte.MAX_VALUE;

  public StableBloomSet(int d,byte C,int m, int[] seeds)
    throws IOException {
      super(m,seeds);
      this.d = d;
      this.C = C;
  }
```

在该算法的理论分析中，是随机选择d个计数器来递减的。但是，这非常慢，而根据分析，唯一的要求就是每个计数器每轮有d/m的可能被递减。因此，算法发明者的做法是，随机选择一个点k，然后将其后的d - 1个计数器递减，如下列代码所示：

```
private final MersenneTwister random = new MersenneTwister();
public void decrement() {
  int k = random.nextInt();
  for(int i=0;i<d;i++)
    if(counters[(k+i) % m] > 0) counters[(k+i) % m]--;
}
```

在更新步骤中，首先调用该递减函数来使过滤器“老化”，然后将相应的计数器置为最大值，而不是简单地递增计数器，从而用新元素更新过滤器：

```
public boolean add(E arg0) {
  decrement();
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int j = hashes[i].hash() % m;
      counters[j] = C;
    } catch(IOException e) {
    }
  }
  return true;
}
```

集合成员的计算和过滤器大小估计的实现则保持不变。

该方法的缺点是，它引入了假阴性错误的可能。为了减少实际情况下假阴性的概率，最初论文的作者对d和c的不同取值进行了大量实验研究。总的经验法则是，让c尽量小，因为较大的c需要较大的d，这会在每次更新时引入更多步骤，从而使整个算法变慢。确定了计数器的大小之后，可以对略图进行实验性测试，来设置递减的元素数量。

10.4　Distinct Value略图

Bloom Filter是一种非常灵活的算法，适合对集合成员和基数进行近似。如果只需要近似集合的基数，那就可以使用特定的Distinct Value（DV）略图来改进Bloom Filter的错误率，并减少内存占用空间。多年以来，人们开发了许多DV略图算法。这些算法通常可以归为两类，本节对其逐一讨论。

10.4.1　Min-Count算法

第一类的代表是依赖于次序统计（order statistics）分布的Min-Count算法（Frédéric Giroire，“Order Statistics and Estimating Cardinalities of massive Data Sets，”Discrete Applied Mathematics，2009，Vol 157 Issue 2（January），406-427）。为了理解该算法的原理，需要回想一下前面对哈希函数的讨论。一个优秀的哈希函数获得一个输入值，将其映射到包含2^p 个值的空间，这些值在[0，2^p - 1]上均匀分布。

通过该均匀分布，可以得到一个输入哈希到小于x的值的概率为$x/2^p$。假设观察到n个各不相同的哈希值S_1 ，…，S_n ，设X是这些值中最小的。X不大于某个值x的概率就是1减去S的所有值均大于x的概率：

$$P(X \le x) = 1 - ((1 - P(S_1 < x)) * (1 - P(S_2 < x)) * \ldots * (1 - P(S_n < x)))$$

由于所有S值的分布都是相同的，因此可以简化为：

$$P(X \leqslant x) = 1 - (1 - P(S\neg_{I} < x))^{n}$$

使用该公式来计算以n表示的X的期望值：

$E[X] = 2^p / (n + 1)$

如果M是插入n个元素之后所观察到的最小值，根据第9章的Delta方法，就可以得到E[1/X]近似为1/E[X]。由此可以求出以M表示的n的解：

$n = 2^p / M - 1 \sim 2^p / M$

这个近似值很容易计算，但当M等于0时就有明显的问题，因为此时Delta方法的计算是未定义的。此外，该近似的方差往往要比要求的高。为了解决这些问题，Min-Count算法引入了两项技术来改进近似误差，并解决当M变得非常小时近似值过大的问题。

首先，对于第二个问题，该算法是用非常简单的方式解决的。我们不再直接记录最小值，而是记录k个最小值，然后在第k个最小值上进行计算，这个值的期望肯定是有定义的。这需要k倍的存储空间，但k通常是一个比较小的数。实际上，该算法的简单版本在k = 3时效果最好。

对于第一个问题，为了改进算法的误差，其中一个方法就是使用与Bloom Filter相同的方法，引入m个不同的哈希函数来生成m个相互独立的数据流。取这m个相互独立的流的数学平均，就可以得到与单个流相同的期望，而标准差则正比于$1/\sqrt{m}$，这样就改进了标准差。使用m个独立哈希函数的计算代价比较大，因此Min-Count使用了名为随机平均（stochastic averaging）的技术，该技术是Flajolet在1985年提出来的，最初用来模拟相互独立的流。

为了模拟相互独立的随机流，哈希值的前b位用来定义2^b个寄存器，其中b通常要小于哈希函数中所用位数量的一半。如果存储3个最小值的每个寄存器都被规格化为[0，1]之间的数，那么基数的估计值就变成：

$$N = 2^*（1/M_1）+（1/M_2）+ \ldots +（1/M_m）$$

该估计值的标准差为$\sqrt{m}$。让我们举一个具体的例子，32位的哈希函数和65536个寄存器需要384k比特的存储空间。与之相比，如果使用抽样方法减少存储空间，同样使用32位哈希，在相同的存储空间中只能存储98000个值。当不同值的个数很少，而数据流规模很大时，这显然是不够的。

10.4.1.1 实现Min-Count略图

与Bloom Filter的实现类似，Min-Count的实现使用标准的Java Set接口，并使用序列化元素。该实现还是用第3个最小值来避免上一节中提到的问题。使用参数b来定义要用的位数，并定义2b个寄存器。该类的初始化代码如下所示：

```
public class MinCount<E extends Serializable> implements Set<E> {

  int   b;
int[]             M;
SerializableHasher hash = new SerializableHasher();
ObjectOutputStream out;

public MinCount(int b) throws IOException {
this.b = b;
int m  = (int) Math.pow(2, b);
this.M = new int[3*m];
out    = new ObjectOutputStream(hash);
reset();
  }
```

Add（）方法的实现很简单。只需要将输入值哈希，然后更新那3个最小值：

```
public boolean add(E arg0) {
  try {
hash.reset();
out.writeObject(arg0);
out.flush();
int h = hash.hash();

int m = (h >>> (32-b));
int v = (h << b) >>> b;

if(M[m] > v) {
M[m+2] = M[m+1];
M[m]   = v;
} else if(M[m+1] > v) {
M[m+2] = M[m+1];
M[m+1] = v;
} else if(M[m+2] > v) {
M[m+2] = v;


}
return true;
} catch(IOException e) {
return false;
}
}
```

现在剩下的唯一的事情就是实现size（）方法，以生成基数的估计值：

```
public int size() {
  double range    = Math.pow(2,32-b);
  double estimate = 0.0;
  for(int i=0;i<M.length/3;i+=3)
    estimate += range/(double)M[i+2];
  return (int)(2.0*estimate);
}
```

10.4.1.2　计算集合的相似度

利用名为Min-wise哈希 的技术，Min-Count略图也可以用来计算两个

集合间相似度的近似值。其基本思想是，计算集合间Jaccard相似度度量的近似值，这个度量我们在本章前面10.3.4节的“对欺诈网站的实时识别”边栏中使用过。

一个关键的观察是，两个集合的最小哈希值相等的概率等于这两个集合交集的大小除以这两个集合并集的大小。这恰恰就是10.3.4节“对欺诈网站的实时识别”边栏中的Jaccard相似度度量。

与多数略图类似，只使用一个哈希函数的近似是非常粗略的。为了克服这个缺点，该算法最早的变体使用多个哈希函数，这与Bloom Filter很相似。当然，这样做代价很高，至少需要400个哈希函数才能达到5%的错误率。后来的变体使用一个哈希函数，然后对哈希函数进行二次抽样，生成多个流，这就类似于Min-Count略图。不论哪种方法，Jaccard相似度都是两个集合的寄存器的匹配比例，并且这两个寄存器必须是用相同的哈希机制得到的：

```
public double distance(MinCount other) {
  double x = 0.0;
  for(int i=0;i<M.length;i++) {
    if(other.M[i] == M[i]) x += 1.0;
  }
  return x/(double)M.length;
}
```

10.4.2　HyperLogLog算法

HyperLogLog是LogLog和SuperLogLog等一长串算法中最近的一个，它是由Flajolet（我们在第10.4.1节中提到过，他发明了著名的随机平均法）在2007年提出的。这些算法都使用位模式的期望分布来估计基数，而

不是像Min-Count和其相关算法那样使用次序统计量。在HyperLogLog中，寄存器中记录的是最近一次运行时哈希值的前导0（leading zeros），而不是最小值或最大值。

为了理解该算法的工作原理，回想一下我们曾假设哈希函数是均匀分布的。这意味着对k位的哈希值来说，它每一位的值都是等概率抛硬币的结果。因此，第一位为1的概率是50%。换句话说，如果观察到两个不同的哈希值，可以预期其中一个会以1打头。同样的逻辑也可以应用到前导0的其他模式。以01打头的哈希值的出现概率为25%，因此可以预期，在4个不同的哈希值中它会出现1次。一般来说，以r个0打头的哈希模式的概率是$p = 2^{1-r} 2^{-1} = 2^{-r}$，可以预期，在$1/p = 2^r$个不同值中，能观察到该模式1次。

与Min-Count算法类似，HyperLogLog使用随机平均来提高性能。同样会将流划分为2^b个流，其中b取值范围为4到16。每个寄存器存放第一个1的最高位置，为每个流提供$2^{M[i]}$的比率。由于这些都是比率，因此算法计算一个规格化的调和平均值，这在计算平均比率时很有用：

$$N = a(b) \times m^2 / (2^{-M[1]} + 2^{-M[2]} + \ldots + 2^{-M[m]})$$

这里的函数a（b）是调整因子，Flajolet用它来修正估计偏差。该函数是一个积分函数，不过，对于b的有效范围，可以利用简单的公式得到函数值。当b为4、5和6时，可以将a（b）分别视为常数0.673、0.697和0.709。当b处于7和16之间时，$a(b) = 0.7213/(1 + 1.079/2^b)$。

当基数处于“中间”区域时，这种估计方法效果不错。当基数非常大或

非常小时，还可以进一步修正。

当原始的估计值小于$5 \times m/2$时，很可能有寄存器没有用到。用V代表未用到的寄存器的数量，如果V大于0，则N最好由下列公式来估计：

$$N^* = m \times \log(m/V)$$

这实际上和名为线性计数（Linear Counting）的另一个算法的结果很相似。

类似地，当基数非常大时，会开始引入哈希冲突，从而导致低估各项的计数。当原始的估计值超过$2^{32}/30$时（此时基数约为1.43亿），就要进行修正：

$$N^* = -2^{32} \times \log(1 - N/2^{32})$$

使用上述修正，基数估计的相对误差近似为$1.04\sqrt{m}$。

10.4.2.1 HyperLogLog的实现

与使用Min-Count相比，使用HyperLogLog最明显的好处就是，寄存器需要的存储空间要小得多。该算法不需要像Min-Count那样存储最小哈希值，而只需要存储前导0的最大数量。对于32位哈希函数来说，这个数量最多只能为32，只需要5位就可以存储起来。而对于相同位数的哈希函数，Min-Count算法则需要32位。这样，如果使用HyperLogLog，会让存储空间一下子缩减85%。当然，为每个寄存器分配8位通常要更容易些，即使这样，也减少了75%的存储空间需求。

只有当所用寄存器的数量处于2^4和2^{16}之间时，HyperLogLog算法才是良好定义的，因此在创建对象时，要严格保证寄存器的数量满足该范围：

```
int m;

public HyperLogLog(int b) throws IOException {
  output = new ObjectOutputStream(hash);
  this.b = b;
  if(b < 4) b = 4; else if(b > 16) b = 16;
  m = 1 << b;
  M = new byte[m];
  for(int i=0;i<m;i++) M[i] = 0;
  V = M.length;

  switch(b) {
    case 4: alpha = 0.673*m*m;break;

    case 5: alpha = 0.697*m*m;break;
    case 6: alpha = 0.709*m*m;break;
    default: alpha = (0.7212/(1+1.079/m))*m*m;
  }
}
```

要注意的是，这里对基数计算中使用的某些常数进行了预计算，以减少后续的处理。

接下来是add（E arg0）方法的实现。由于我们预计会频繁用到集合的大小，因此该实现直接维护了寄存器的和。这样就使基数估计减少了O（m）的计算量。因为同样的原因，该实现还维护了空寄存器的数量，用来对小基数的估计进行修正。这样，该实现就要比Min-Count的add方法稍复杂一些：

```
public boolean add(E e) {
  try {
    hash.reset();
    output.writeObject(e);
    int x = hash.hash();
    int j = x >>> (Integer.MAX_VALUE - b);
    x = (x << b)|(1 << (b-1)) + 1;
    int r = Integer.numberOfLeadingZeros(x)+1;

    if(M[j] == 0) {
      V--;
      M[j] = (byte)r;
      return true;
    } else if(r > M[j]) {
      Z -= Math.pow(2, -M[j]);
      Z += Math.pow(2, -r);
      M[j] = (byte)r;
      return true;
    }
    return false;
  } catch(IOException ex) {
    return false;
  }
}
```

最后，size（）方法的实现使用基线值（baseline value）Z（Flajolet把它叫做示性函数（indicator function）），来生成基数的修正估计值：

```
public int size() {
  double N = alpha/Z;
  if(N <= 5.0*m/2.0 && V > 0) {
    return (int) (m*Math.log((double)m/(double)V));
  } else if(N > Math.pow(2, 32)/30.0){
    return (int) (-Math.pow(2, 32)
    *Math.log(1.0 - N/Math.pow(2, 32)));
  } else
return (int)N;
}
```

10.4.2.2 对HyperLogLog算法的改进

HyperLogLog算法原本被设计用来处理的场景是，要度量的集合预计拥有十亿级别的基数。基于这个近似的基数，当基数的原始估计值偏大时，HyperLogLog的发明者在集合大小计算的实现中加入了对原始估计值N的修正。该修正用来抵消不断变大的哈希冲突概率的影响，并对超过1.4

亿的基数确定估计值。

对多数用户来说，这个基数范围绰绰有余。在单个网站上运行的应用，他们所处理的基数通常是几百万。处理像在线广告这样的多个网站的应用，他们面对的基数可以达到几亿，极端情况下可能达到十亿。

不过，Google公司日常处理的基数就超过十亿。为此，Google对HyperLogLog（Google称其为HyperLogLog＋＋）进行了大量修改，并将其成果整理成论文（该论文还提到，Google也曾是Min-Count算法的重量级用户），于2013年年初发表。

为了使用HyperLogLog估计非常大的基数，Google将32位哈希函数换成了64位哈希函数，并且使用不为外人所知的专有哈希函数。不过，我们知道的是，在该实现中使用了Murmur哈希函数的64位变体。

这种修改有两个方面的影响。第一，需要更多的空间来使寄存器的存储空间最小化，因为现在需要6位，而不是原来的5位。在上一节给出的实现中，使用了8位来表示计数器，这不是最佳的，因此这样的改变没有什么效果。

第二，哈希冲突的概率要比原来的32位哈希函数小得多，因此对于实际基数不再需要大范围的修正。如果基数达到了比较高的水平，该论文的作者建议，只需要让哈希函数更大，就很可能获得更好的估计值。将哈希函数的大小加倍，这只增加了1位的存储空间。在论文给出的实现中，可以使用256位的哈希函数，而不需要任何额外的存储空间。

不过，我们可能仍然需要对小基数进行修正，这时使用的是完全不同

的算法！为了在基数较小时改善偏差，Google对于熟知的基数进行了深入的实证调查。由此得到了一系列插值参考点，可以在基数较小时使用。因此，size（）方法的实现就变成：

```
public int size () {
  double N = alpha/Z;
if(N < 5.0*m) N -= estimateBias(N);
double H = (V > 0) ?
-Math.pow(2, 32)*Math.log(1.0 - N/Math.pow(2, 32)) : N;
  return (int)(H < threshold(b) ? H : N);
}
```

两个函数threshold（）和estimateBias（）的值是由Google经验性确定的，参见http://goo.gl.iU8Ig 。

对各不相同的网站实时访问者的数据透视表

第6章中使用了压缩位图来高效地存储数据，从而为网站的页面浏览、点击等活动创建数据透视表。这颇为有趣，但比起讨论页面浏览活动，讨论网站的访问者往往要更加有趣。

使用压缩位图来跟踪各不相同的用户是不可行的，因为新元素的加入是只能追加的操作。因此需要一个单调递增计数器。在原来的例子中，选择数据到达的顺序作为该计数器。相反，在某个时间段内各不相同的访问者的顺序不是固定的，甚至很可能会在该时间段内多次出现。

可以用HyperLogLog略图替代压缩位图，并使用容斥原理来对数据透视表所需要的交集大小进行估计。首先，让我们回想一下网站页面浏览的输入数据的格式：

```
timestamp,user id,feature1:value,feature2:value,...,featureN:value
```

每条输入记录都包含一个用户ID，用作输入到HyperLogLog略图的元素。还包含一些特征元素，这些特征元素定义了用户的相关人口统计信息，或他们所访问页面的其他相关信息（例如Web站点或产品页面）。

HyperLogLog基本上会为每个特征生成略图，这些特征就是需要维护的值的组合，其中包含各不相同用户ID及其近似数量。要对任意两个特征A和B建立数据透视表，只需要获取特征A的所有可能取值的略图，即特征A：值1，特征A：值2，...，特征A：值N，以及特征B的所有可能取值的略图，即特征B：值1，特征B：值2，...，特征B：值N。然后用容斥原理计算每一对特征A：值X，特征B：值Y的交集：|特征A：值X| + |特征B：值Y| - |特征A：值X与特征B：值Y的并集|。

只要每个特征可取的不同值的数量较小，这个过程就可以交互进行。如果每个特征可取值的数量很大，最好的方法是使用Count-Min略图来找出最常见的特征值，以控制在数据透视表用户界面中显示的特征值数量。

10.5 Count-Min略图

最后要讨论的略图算法是广泛采用的Count-Min略图。从本质上说，该算法近似了一个映射（也成为多集），其中各不相同的值作为键，元素出现的次数作为值。在Count-Min略图最简单的实现中，它支持对映射进行近似查找，称为点查询（point guery）。当这些各不相同的值有某种内在的顺序时，可以将该略图视为对直方图的近似（也被称为频率表）。

只需要多做一点工作，Count-Min略图还可以支持范围查询，从而得到处于范围[a，b]中的元素的近似频率。支持范围查询后，还可以对取值范围使用二分查找，来估计次序统计量。本章后面会对此做详细讨论。

注意　本章讨论的多数略图只支持向略图中加入元素，而不支持从略图中删除元素。最早提出Count-Min的论文的作者将这种模型称为收银机模型（Cash Register model），该模型假设所有的频率都是非负的。Count-Min略图支持该模型，但也支持作者所称的十字门模型（Turnstile model）。在十字门模型中，可以从集合中删除元素，因此值可以是负的。本章主要关注收银机模型，因为这种模型在流数据中更为常见，但十字门模型对于Count-Min略图也是有意义的。

Count-Min略图与Counting Bloom Filter在实现上很相似。实际上，已经有人证明了，Counting Bloom Filter的一个名为Spectral Bloom Filter的扩展等价于Count-Min略图。与Counting Bloom Filter类似，Count-Min略图用一个计数器数组作为寄存器。两者的主要区别在于，在后者的k个哈希函数中，每一个都有自己的m个寄存器集合。这就定义了一个寄存器

值的二维矩阵，其中一个哈希函数只负责更新特定的一行。K的值通常被视为略图的“深度”，m的值则被视为略图的“宽度”。

向该略图加入元素也与Counting Bloom Filter很相似。首先将该元素输入给每个哈希函数hi（x），从而计算哈希值，然后再对哈希值进行常见的取模操作，将其映射到m来确定寄存器j。对每个哈希函数，该算法增加的是位于mij的寄存器。

10.5.1　点查询

计算元素的近似频率也非常简单。应用每个哈希函数之后，算法计算min（m_{1j} ，…，m_{kj} ），将这个值作为该元素被加入集合的近似次数。在现实情况中，该算法效果很好，这可能有点令人吃惊。其实这里的原因很简单，它与其他略图在一个方面非常相似，即都依赖于这个事实：尽管单个哈希函数冲突的概率可能太高了，但多个哈希函数发生冲突的概率是足够小的，因此很好用。

显然，在收银机模型下，该方法倾向于对值的近似频率过高估计。其他元素也可能（当进入略图的元素足够多时，这是必然的）会增加相同的计数器，这与Bloom Filter很相似。那么，Count-Min略图对计数的估计误差是多大？

在最早的Count-Min论文中，作者证明了，只要满足两个条件，对元素的估计值处于其真值x和上界x + em（m是进入映射的元素数量）之间的概率就大于1 - d。第一个条件是，略图的宽度为2/e；第二个条件是，略

图的深度为（log 1/d）/log 2。因此，如果估计误差为元素总数5%的概率为99%，则该Count-Min略图的宽度为40，深度为7。而对深度为8，宽度为128的Count-Min略图，其相对估计误差为1.5%的概率大约等于99.6%。如果使用32位的计数器，Count-Min略图需要的存储空间与b = 12时的HyperLogLog略图相同。

10.5.2　Count-Min略图的实现

Count-Min略图不实现Set＜E＞，而是实现Java类Map＜E，Long＞，只不过加上了一些限制。

```
public class CountMinSketch<E extends Serializable>
  implements Map<E,Long> {

  int width = 0;
  int depth = 0;
  byte[]  m;

  SerializableHasher[] hashes;
  ObjectOutputStream[] outputs;

  protected void initialize(int m,int[] seeds) throws IOException {
    hashes = new SerializableHasher[seeds.length];
    outputs= new ObjectOutputStream[seeds.length];
    for(int i=0;i<seeds.length;i++) {
      hashes[i]  = (new SerializableHasher()).seed(seeds[i]);
      outputs[i] = new ObjectOutputStream(hashes[i]);
    }
  }


  public CountMinSketch(int width,int[] seeds) throws IOException {
    this.width = width;
    this.depth = seeds.length;
    m = new byte[width*depth];
    initialize(m.length,seeds);
  }
```

更新步骤只需要增加相应的寄存器，并将最小的寄存器值作为对频率

的估计值即可。下列代码实现了对略图的特定量的更新，这在两个略图合并时很有用。另外，还对查找过程进行了简化。代码如下：

```
public Long increment(Object arg0,long amount) {
  int min = Integer.MAX_VALUE;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int h = hashes[i].hash() % width;
      m[i*width + h] += amount;
      if(m[i*width + h] < min) min = m[i*width + h];
    } catch(IOException e) {
    }
  }
  return (long) min;
}

public Long increment(Object arg0) { return increment(arg0,1); }

public Long get(Object arg0) {
  return increment(arg0,0);
}
```

10.5.3 Top-K和“Heavy Hitters”

有一个常见的Count-Min应用，那就是维护频繁项的列表。该列表有两个基本的形式，即top-K列表和所谓的Heavy Hitters（高命中率查询）列表。前者就是数据流中最常见的k个元素的列表，后者则是频率高于某个预定值f的元素的列表。

两者的基本实现都用Count-Min略图来存储频率，用堆（heap）数据结构来保存最高值。在Java中，可以用PriorityQueue和适当的Comparator来实现相应的堆，如下面的例子所示：

```
public class SketchComparator<T extends Serializable>
  implements Comparator<T> {

  CountMinSketch<T> sketch;

  public SketchComparator(CountMinSketch<T> sketch) {
    this.sketch = sketch;
  }

  public int compare(T o1, T o2) {
    return (int)(sketch.get(o1) - sketch.get(o2));
  }

}
```

要实现top-K列表，只需要为输入元素更新略图，然后将其加入优先级队列。如果该队列超过k（最多只能包含k + 1个元素），就将最小元素删除，从而使队列大小变回k：

```
public class TopList<E extends Serializable> {
  CountMinSketch<E> sketch;
  PriorityQueue<E>  heap;


  int               k       = Integer.MAX_VALUE;

  public TopList(int k,CountMinSketch<E> sketch) {
    this.sketch = sketch;
    this.k      = k;
    this.heap   = new PriorityQueue<E>(k+1,new
SketchComparator<E>(sketch));
  }

  public void add(E element) {
    sketch.increment(element);
    heap.add(element);
    while(heap.size() > k) heap.remove();
  }

}
```

Heavy Hitter的实现几乎是相同的，唯一的不同点是，当元素不满足频率要求时，要将该元素删除。这里，同样要为目前为止处理的所有元素维护一个计数器：

```
public class HeavyHitters<E extends Serializable> {

  CountMinSketch<E> sketch;
  PriorityQueue<E>  heap;
  double            f = 1.0;
  int               N = 0;

  public HeavyHitters(double f,CountMinSketch<E> sketch) {
    this.sketch = sketch;
    this.heap   = new PriorityQueue<E>(11,
      new SketchComparator<E>(sketch));
    this.f      = f;
  }

  public void add(E element) {
    N++;
    sketch.increment(element);
    heap.add(element);
    while(heap.size() > 0 &&
          f <= (double)sketch.get(heap.peek())/(double)N)
      heap.remove();
  }

}
```

还有另外一种实现方法，该方法不是在每次插入元素时裁剪队列，而是一直等到队列因为某种原因被查询才进行裁剪。

网站中的最热门商品

在Amazon.com或Etsy.com等商业网站中，通常都有展示最热门商品的侧边栏。这些网站的每个部门往往都有上千件产品。为了实时维护top-K列表，网站就必须以便捷的方式维护每件商品的当前购买量。对于Amazon.com来说，由于拥有那么多的硬件设备，因此已经开始把硬件租赁作为副业。故而对于它来说，维护top-k似乎不成问题，但对于相对较小的特征来说，还是太昂贵了。

使用上一节的top-K列表方法，只需要使用几K字节的RAM存储，就可以便宜地实现近似的top-10最热门商品。如果要提高准确度，可以为网站

的每个部门都维护一个top-K列表，这样各页面都有唯一的top-K列表。

10.5.4　范围查询和分位数查询

许多时候，要存储的数据都具有某种天然的顺序。此时，我们会很自然地将数据看作服从某种经验分布（或是直方图，这取决于具体学科）。这类数据的例子包括，对网站上特定广告的支付价格（以美分为单位），或者用户在给定页面上停留了几秒才进行操作。

在这些情况下，我们感兴趣的问题通常都涉及某种范围查询：有多大比例的用户在该页面的停留时间小于10秒？该广告售价的中位数是多少？对第一个问题的原始解法是，简单地将从0秒到10秒的点估计加起来，然后得到总的频率。而要回答广告价格中位数的问题，就得从0.01美元开始计算频率，直到点估计的和近似等于当时所看到元素总量的50%为止。

这样的原始方法存在两个方面的问题。第一个方面的问题是需要的运算量可能很大，对于上文中的第二个问题（中位数问题）这是显而易见的。如果价格中位数是15美元，那么就需要1500个点估计，每个都要进行k个哈希函数的计算。第二个方面的问题是，这些点估计都有一定的误差，这个问题就没那么明显。如果将这些点估计相加，正如第9章所讨论的那样，每次相加都会导致误差增大。即使是最乐观的情况，预期的中位数计算误差也是任何单个点估计的1500倍。

有一个不那么原始的方法，就是找到某种方式来减少计算范围频率所需要的点查询的数量。一种方法是使用多个Count-Min略图，以不同的粒

度存储全部数据。第一个略图是基本略图，用来存储点查询。第二个略图将第一个和第二个元素合并到一个计数器，从而将粒度减半，依此类推。第三个略图将第二个略图的前两个元素合并起来，因此其第一个元素就是最初略图的第一、二、三、四元素的计数值。这样，粒度会一直减半，直到最终的略图只包含两个元素。

使用这种方法，任何范围查询都可以被划分为最多$2\times \lg N$个区间，其中N是值域的大小。例如，如果值域是32位整数空间，那么计算任何范围查询最多只需要64个子区间。与原始方法的O（n）计算时间相比，这是一个不小的进步。

10.5.4.1　二进区间

上一节给出的区间称为二进区间（dyadic intervals）。二进区间由两个参数定义：level参数和start参数：

```
public class DyadicInterval {
  public int level = 0;
  public int start = 0;

  public DyadicInterval() { }
  public DyadicInterval(int level,int start) {
    this.level = level;
    this.start = start;
  }
```

这两个参数将区间的起点和终点定义为$[2^{level}\times start，2^{level}\times（start+1）-1]$：

```
public int min() { return (1 << level)*start; }
public int max() { return (1 << level)*(start+1) - 1; }
```

然后寻找处于当前区间范围内的最大的二进区间，从而递归地计算子

区间。将区间加入到最终的输出，并可能再生成两个区间：左区间（字前区间以左）和右区间（字前区间以右）。再以同样的方式递归地将这些区间划分为子区间，直到所有点都被包含到一个范围中：

```
public static List<DyadicInterval> createIntervals(int a,int b) {
    ArrayList<DyadicInterval> output = new ArrayList<DyadicInterval>();

    Stack<Integer> left = new Stack<Integer>();
    Stack<Integer> right = new Stack<Integer>();
    left.push(Math.min(a, b));
    right.push(Math.max(a, b)+1);

    while(left.size() > 0) {
      int l = left.pop();
      int r = right.pop();

      for(int k = 32;k>=0;k--) {
        long J = (1L << k);
        long L = (int)(J*Math.ceil((double)l/(double)J));
        long R = L + J - 1;
        if(R < r) {
          //Segment fits in the range
          output.add(new DyadicInterval((int)k,(int)(L/J)));
          if(L > l) {
            left.push(l);right.push((int)L);
          }
          if(R+1 < r) {
            left.push((int)(R+1));right.push(r);
          }
          break;
        }
      }
    }
```

10.5.4.2 基于Count-Min的范围查询的实现

覆盖32位整数空间的Count-Min略图的实现使用了从0到31共32个不同的level，每个都有相同的宽度和深度：

```
public class CountMinRange {
  ArrayList<CountMinSketch<Integer>> sketches =
    new ArrayList<CountMinSketch<Integer>>();
  long N = 0;
  int  min = Integer.MAX_VALUE;
  int  max = Integer.MIN_VALUE;

  public CountMinRange(int width,int[] seeds)
    throws IOException {
    for(int i=0;i<32;i++)
      sketches.add(
      new CountMinSketch<Integer>(width,seeds));
  }
```

在更新步骤中，将计数值加入到每个不同level对应的start参数：

```
public void increment(int value,long n) {
  if(value < min) min = value;
  if(value > max) max = value;
  for(int i=0;i<32;i++) {
    sketches.get(i).increment(value/(1 <<
  }
  N += n;
}
```

范围计算要稍微复杂一些。首先，要计算二进区间的对应集合，然后将每个level的相应元素计入总和。每个范围查询最多计入每个level的两个元素：

```
public long range(int a,int b) {
  long count = 0;
  for(DyadicInterval d : DyadicInterval.createIntervals(a, b)) {
    count += sketches.get(d.level).get(d.start);
  }
  return count;
}
```

分位数（quantile）（例如中位数）是通过对空间的二分查找得到的。确切的概率不太可能位于集合中，但算法会返回与相应的分位数最接近的值。

```
public int quantile(double q){
  if(q == 0.0) return min;
  if(q == 1.0) return max;

  int lo = min;
  int hi = max;
  while(lo <= hi) {
    int mid = lo + (hi - lo)/2;
    double val = frequency(min,mid);
    if(val < q) hi = mid - 1;
    else if(val > q) lo = mid + 1;
    else return mid;
  }
  return lo; //Return the nearest value
}
```

10.6 其他应用

本章介绍了具有相似特性的一类降维工具。这些特性可以从本质上归纳为，存储和更新不直接取决于要处理的元素数量。具备了这样的性质，这可以使流数据应用保持对处理时间的控制，而处理时间正是高性能、低延迟应用的关键。

尽管本书主要关注流应用，但这些算法都是普遍适用的。这些数据结构尤其经常在Map-Reduce应用中大显身手，因为进入Map和Reduce阶段的数据具有与流数据非常相似的行为。

例如，在过滤应用中，在Reduce步骤进行最后的收尾工作前，往往会在Map步骤中使用Bloom Filter，对数据进行粗略的过滤。有一个此类应用称为“归因”（attribution）过程。在该类应用中，用户可以由唯一的标识符来确定，他们先是参与了大量事件，之后可能参与了一个我们感兴趣的事件，这里将该事件称为“转化”。归因过程感兴趣的是，在最终的转化事件前，发生于某个窗口中的事件。由于多数用户是不会转化的（低至个位数的转化百分比是正常的），因此就可以在归因过程的Map-Reduce作业的Map步骤中，使用包含未转化用户ID的Bloom Filter。即使有10%的高误差率，这种方法仍然往往能使进入Reduce阶段的数据量减少80%至90%。

10.7 总结

流数据处理有一个主要的挑战，那就是处理速度要跟得上事件数量的快速增长。虽然发明了高性能的固态硬盘（SSD），但数据通常还是不得不存储在主存储器（RAM）中，以达到可接受的性能。如果要存储的数据比较简单，例如和值或平均值，那么这倒也不成问题。

当要存储的数据更复杂一些时，例如流中不同值的数量，这就可能是个问题。如果尝试直接存储这些数据，就会出现存储需求正比于数据流大小的情况，需要的存储空间很快就会超出内存的可用容量。

本章介绍了一系列方法，用来存储集合和集合的大小这类特定值。我们通过由应用而不是数据控制内存占用率，来确保数据的存储不超过内存的限制。这类技术的缺点是，它们向所计算的值引入了估计误差。在某些情况下，我们可能无法容忍这些误差。但误差是存储量的函数，因此可以通过应用将其调整到可接受的水平。

第11章　流数据的应用

对许多企业来说，聚集计算和可视化就是流数据分析的终点。创建仪表板应用程序，将聚集计算的结果用图形可视化，再生成分析报告。接下来，还有什么要做的？答案通常是，让“决策者”把握系统所监控的东西的脉搏。让我们对这句话稍作解释。这个答案隐含的意思是，系统的作用是将信息提供给人类决策者，以使他们可以采取对自己有利的方式来影响系统，或者对系统中出现的不利变化做出反应。

但是，这些系统都运行于实时环境中，而我们人类是无法做到实时响应的生物。我们要吃饭、睡觉，并且通常还要做别的事，不可能整天盯着不断更新的仪表板。那我们怎么才能对这些信息保持关注呢？对于关键业务来说，解决方法通常是，让一拨人轮流值班来监控系统。

对于比较小的系统，这种方式非常好，但即使是像发电厂这样合理规模的系统，也会很快突破“人海战术”的极限。为此，人们在实时系统中引入了自动化元素，例如告警和自动关闭。

这些自动化的系统正是本章关注的焦点，即通过使决策者聚焦于异常行为，或将时刻变化的决策支持过程完全自动化，来协助决策者工作。这说起来容易做起来难。但过去的时间已经证明，在提供至少有限的自动化能力方面，许多方法已经获得了成功。

不论是将决策过程自动化，还是将向系统操作员自动发出异常数据告警，都需要系统具备某种模型，该模型要使系统能预测自己的行为。本章

的前两节都使用了第9章介绍的“用抽样近似流数据”的概念，来讨论建模过程。建模过程称为“拟合”，是统计学和机器学习应用的基本任务。本章介绍了经典建模背后的一些方法，这些方法通常是“离线”进行的。然后我们将这些方法应用到实时数据中。我们还介绍以“在线”方式将模型与流数据拟合的几种方法。

建立模型之后，就需要建立在这些模型上的应用。实时数据最流行的两类应用是监控应用和优化应用。毋庸置疑，它们分别对应前面提到的识别数据中的异常行为和做出即时决策的思想。

监控包含实时数据的一些经典话题。前面几章我们详细讨论监控数据的采集以及监控结果的可视化，剩下的内容就是异常事件的识别。11.3节讨论两类异常检测问题。第一类问题发生时，系统进入短暂的异常状态，这时要进行离群点检测。第二类问题发生时，系统进入截然不同的运行状态，这时要进行变化检测。

最后一节讨论互联网世界的一个热门话题：“优化”。其中，网站优化是优化的主流课题，对于所谓的A/B测试有不计其数的优化方法。实际上，多数网站流量监控软件大概都内置有某种A/B测试框架。本章使用了一项名为多臂赌博机（multi-armed bandit）的技术，并结合第一节中的建模方法，以在实时环境中实现优化。

11.1 实时数据模型

要预测系统的行为，就必须有描述系统行为的底层模型。理想情况下，该描述要比所描述的数据简洁。例如，在牛顿物理体系中，建立简单的方程组，用它们来描述力对对象的作用，就足以预测这些对象长时间的运动情况。

基于上述概念，可以将模型分成为两个部分。模型的第一部分是底层系统的行为，描述世界的不同部分之间是怎样交互的，何以导致了我们观察到的行为。可惜的是，很难出现构成模型的所有变量都可观察的情况，因此通常不大可能完全确定模型。不过，可以只基于观察到的变量来构造模型。模型的第二部分就是无法观察的变量所引入的噪声。

这既适用于底层过程也适用于度量机制。可以用对底层过程相同的方式，对用来度量数据的任何工具建模。例如，假设有多个不同的系统采集相同的数据，如果知道每个采集系统的偏差和方差，就可以对这些系统的数据规格化。

为此，本节介绍了在有噪声的情况下，用来确定底层模型的若干方法。由于本书聚焦于实时数据，因此本节的第一部分内容主要关注对时间序列模型建模的一些较简单的方法。（由于其数据采集机制，实时数据天生就是时间序列数据。）这些方法在各种领域的预测过程中应用广泛。

本节的第二部分讨论线性模型，又名回归模型。这类模型是统计学的经典技术，可能也是目前最常用的建模技术。线性模型描述所度量变量和

结果（同样也会被度量）之间的线性关系。线性模型的一个变体，逻辑回归，是对事件发生的可能性建模的主流方法。

当数据本身是高度非线性的，或者数据的结构很难理解时，人工神经网络就成为时间序列数据建模的主流选择。本节最后的内容是神经网络模型基础。神经网络模型容易适应实时问题，因为它的训练循环和预测循环是相同的。

11.1.1　简单时间序列模型

任何实时数据系统本质上都由一个顺序的值序列组成。我们在第8章讨论过，这些值是基于特定的时间“桶（bucket）”或量（quantization）计算出来的，然后才被继续分析。多数这些度量值天然就是带有噪声的，因此时间序列模型尝试通过消除噪声来获取时间序列的真正模型。这通常是通过某种平均技术进行的，本节我们介绍其中几项平均技术。

11.1.1.1　移动平均

时间序列最简单的平均方法是移动平均（moving average）。顾名思义，它是对“滑动窗口”中值的平均，通常用平均计算中使用的时间桶的数量来表示，经常记为MA（k）。例如，对股票市场数据的移动平均往往用天数来表示。

移动平均很容易实现，因为它只需要存储k个不同值。效率起见，通常最好是始终跟踪当前各个值的总和，并根据需要增加或删除值。如果不这样做，那么每次更新移动平均都需要k次运算，而不是最多两次。如下

列代码所示：

```
public class MovingAverage {

  LinkedList<Double> values = new LinkedList<Double>();
  double sum;
  int    k;

  public MovingAverage(int k) {
    this.k = k;
  }

  public double observe(double value) {
    sum += value;
    values.add(value);
    if(values.size() > k) {
      sum -= values.removeFirst();
      return sum/(double)k;
    }
    return Double.NaN;
  }
}
```

需要注意的是，如果没有收到足够的观察数据，移动平均会返回NaN。为了生成有用的观察值，移动平均会丢弃时间序列的前k - 1个值。对移动平均来说，选择适当的窗口大小是一门艺术。在实际数据上，移动平均起到的是低通滤波器的作用，选择太小的窗口会无法去除足够的噪声。反之，太大的窗口会去除数据中的重要信号。图11-1展示了在一些混有少量噪声的模拟正弦波上选择不同移动平均窗口的影响。图中，窗口大小为30或35基本上能最好地恢复真正的信号。

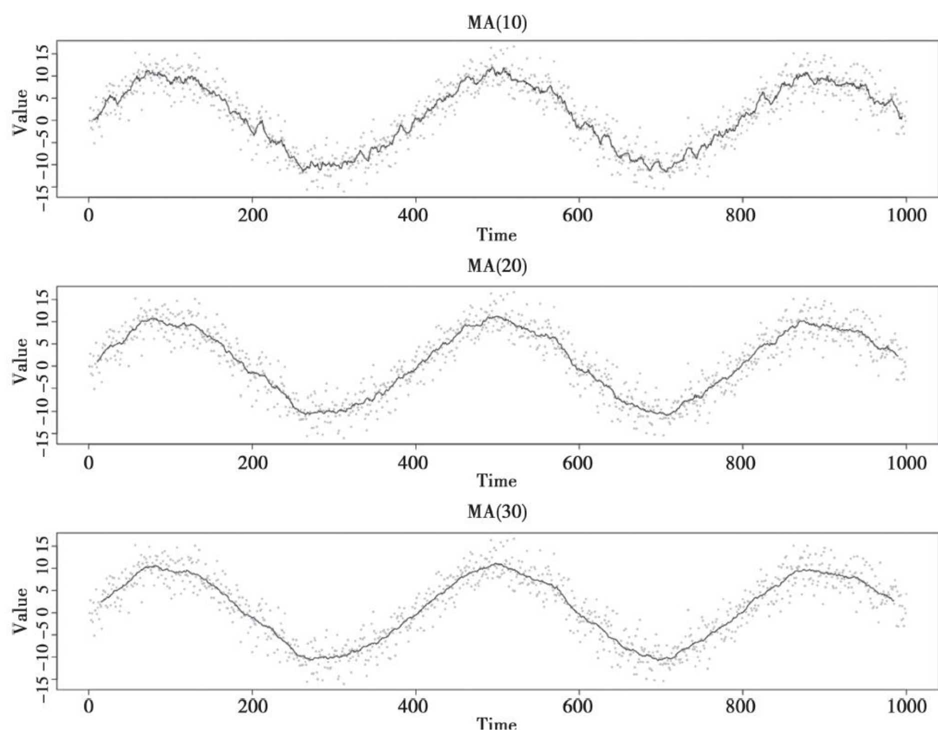


图 11-1

11.1.1.2 加权移动平均

加权移动平均是标准移动平均的扩展，它对窗口中的各个元素使用不同的权重。这些权重的集合称为核（kernel），不同行业对核有不同的标准。常规的移动平均也是一种加权移动平均，它的核对所有值的权重都是1/k。加权移动平均的实现要稍微复杂一些：

```java
public class WeightedMovingAverage {
  double[] kernel;
  double[] values;
  double   kernelSum = 0.0;
  int      k = 0;
  long     N = 0;

  public WeightedMovingAverage(double[] kernel) {
    this.kernel = kernel;
    for(double j : kernel) kernelSum += j;
    values = new double[kernel.length];
  }

  public double observe(double x) {
    values[k++] = x;
    if(k == values.length) k = 0;
    N++;
    if(N < kernel.length) return Double.NaN;
    double y = 0;
    for(int i=0;i<kernel.length;i++)
    y += kernel[i]*values[(k+i) % values.length];
    return y/kernelSum;
  }
}
```

需要注意的一个重要问题是，加权移动平均的计算要比常规移动平均慢很多，对每个观察值都需要k次运算。对于多数离线分析来说，这不成问题，但对于实时环境中的在线处理来说，这可能是个问题。

11.1.1.3 指数移动平均

许多时候，要使用加权移动平均中的核，使最近的观察值比较旧的观察值具有更大的权重。指数移动平均取当前移动平均值与新的观察值的加权平均，从而做到了这一点：

EMA = a*X + （1 - a）*EMA

指数移动平均与加权移动平均并不是完全一样，它有几个优点。第一个优点是，只需要单遍运算就可以获得新的平均值，而不是像加权移动平

均那样需要k次运算。第二个优点是，只需要存储一个值，而移动平均和加权移动平均都需要存储k个值。正因如此，它的实现非常简单：

```
public class ExponentialMovingAverage {
  double value = 0.0;
  double alpha;
  public ExponentialMovingAverage(double alpha) {
    this.alpha = Math.min(alpha, 1.0);
  }

  public double observe(double x) {
    value = alpha*x + (1-alpha)*value;


    return value;
  }
}
```

与常规的移动平均类似，alpha值的选择不像科学，而更像是艺术。最常用的经验法则是将alpha置为2/（k＋1），其中k是在移动平均中使用的值的数量。结果大概是从最近的k个值得到权重的86.5%。如图11-2所示，该结果与标准的移动平均非常接近。图11-2展示了对同一个数据集，由两外两个指数移动平均去除噪声后的结果，其中k分别为20和30。图中还用虚线展示原始的移动平均，以进行互相比较。你可以看到，这两种移动平均的结果十分贴近。由于指数移动平均与原始的移动平均很相近，因此指数移动平均是现实中最常用的算法。

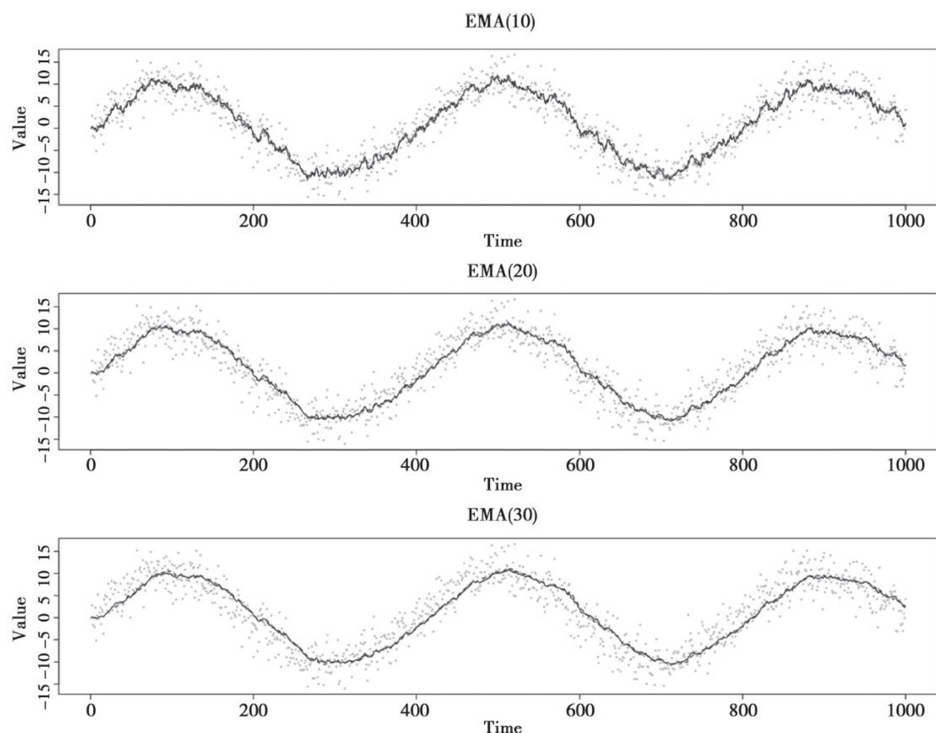


图 11-2

11.1.2 线性模型

线性模型是许多领域中最流行的统计建模方法，这些领域包括经济学、生物学等等（它的首批应用中包括遗传学研究）。这类模型又名回归模型，本书就使用这个名称。也可以称为最小二乘模型，这个名称来自于用来寻找模型系数的技术。

线性模型的基本思想是，输出变量与一个解释变量（explanatory variable）数组x[0]，x[1]，...存在线性关系，它等于解释变量的各分量

与一个系数数组各分量B[0]，B[1]，...的乘积。

```
public class LinearModel {
  double[] B;

  public LinearModel(double[] B) {
    this.B = B;
  }

  public double y(double[] x) {
    double y = 0;
    for(int i=0;i<B.length;i++) y += B[i]*x[i];
    return y;
  }

}
```

对线性模型进行“拟合”意味着，在给定x和y值的前提下，求出合适的B值，其中y可以解释为y的有噪声版本。在最初的公式中，该噪声服从均值为0，方差为s的正态分布（第9章中讨论了正态分布）。首先要根据LinearModel类的y方法，对给定的x得到，然后令y的观察值与的差的平方最小化，从而求得B。

换句话说，我们的目标是找到能返回最小平方和或“最小平方误差”的数组B，平方误差的定义如下：

```
public double error(double[] y,double[][] x) {
  double error = 0.0;
  for(int i=0;i<y.length;i++) {
    double diff = y[i] - y(x[i]);
    error += diff*diff;
  }
  return error;
}
```

该误差又名残差平方和（Residual Sum of Squares，RSS），常用来确定模型与数据的拟合程度。逐个观察误差的各个分量（通常不加平方），可以确定模型是否得到了良好定义。如果数据中存在趋势或周期性

信号，这往往说明模型缺项。

11.1.2.1 简单线性模型

当x数组只包含两个值时（第一个值是常数1，即截距项），B数组的两个值存在一个简单的封闭解。这时，B的第一个值通常称为a，第二个值称为b，y的等式可以化简为下列简单形式：

```
public class SimpleLinearModel {

  public double a,b;

  public SimpleLinearModel(double a,double b) {
    this.a = a;
    this.b = b;
  }

  public double y(double x) {
    return a + b*x;
  }

}
```

最小化的误差同样可以化简为：

```
public double error(double[] y,double [] x) {
  double error = 0.0;
  for(int i=0;i<y.length;i++) error += (y[i]-a-b*x[i])*(y[i]-a-b*x[i]);
  return error;
}
```

只需要少量计算就可以得到，当b等于x和y的协方差与x方差的商时，误差函数取得最小值。当a等于y的均值减去与b的估计值和x均值的乘积时，误差最小。如果将全部数据都放在一个数组中，下列函数可以找到合适的a和b：

```
public void fit(double[] y,double[] x) {
  double sumX = 0.0, sumY = 0.0;
  double sumXY = 0.0, sumX2 = 0.0;
  for(int i=0;i<y.length;i++) {
    sumX += x[i];
    sumY += y[i];
    sumXY += x[i]*y[i];
    sumX2 += x[i]*x[i];
  }
  double n = (double)y.length;
  b = (sumXY - (sumX*sumY)/n)/(sumX2 - (sumX*sumX)/n);
  a = sumY/n - (b*sumX)/n;
}
```

你可能已经注意到，上述函数只需要对数据进行单遍扫描。这意味着它可以用于简单的流式计算，因此对于实时分析很有用：

```
public class StreamingSimpleLinearModel {

  double sumX = 0.0, sumY = 0.0;
  double sumXY = 0.0, sumX2 = 0.0;
  double n = 0.0;

  boolean dirty = true;
  double  a = 0.0,b = 0.0;

  private void update() {
    if(!dirty) return;
    b = (sumXY - (sumX*sumY)/n)/(sumX2 - (sumX*sumX)/n);
    a = sumY/n - b*sumX/n;
    dirty = false;
  }

  public void observe(double y,double x) {
    sumX += x;sumY += y;
    sumXY += x*y;sumX2 += x*x;
    n += 1.0;
    dirty = true;
  }

  public double b() { update();return b; }
  public double a() { update();return a; }
}
```

11.1.2.2　多元线性回归

当有不止一个x变量时，计算B的最佳估计（不用计算截距项）要稍复杂一些，不过仍然是直截了当的。只要满足特定的条件，就可以使用名为

最小二乘法（ordinary leart square）的技术（有时会把最小二乘法作为多元线性回归的同义词），来得到B的封闭解。

要满足的主要条件是，不同的x变量之间没有关联，并且y的标准差不依赖于x的值（只有y的均值随着x值变化）。第一个条件通常很容易达到，可以检查各个x变量之间的关联，然后从等式中去掉两个相关变量中的一个。另一种方法是对x值的矩阵进行正交变换，例如主成分分析。这样就可以生成（根据定义是）不相关的x值。第二个条件通常是假设出来的，而不是要确保的。要在模型拟合之后，检查y的观察值和其预测值的差别，来判断该条件是否得到了满足。

假设这些条件都得到了满足，并且已经将x的值放入k列n行的矩阵X，其中k是不同变量的数量，n是观察次数。那就可以通过下列表达式找到使平方误差的均值最小的向量B：

$$B = (X^T X)^{-1} X^T y$$

该公式是线性方程组的解，这种方程组又名正规方程。可以使用Apache Commons Math库等线性代数库来直接求解。但是，直接求解可能存在数值稳定性方面的问题，因此多数实现使用其他技术。有许多选择，其中最常见的是使用QR分解。QR分解指的是，任何矩阵A都可以用两个矩阵Q和R的乘积来表示，其中Q是正交矩阵（与前面提到的类似），R是上对角矩阵（通俗地讲，矩阵中一半的值都是0）。正交矩阵很特殊，因为$Q^T Q = I$，其中I是单位矩阵。单位矩阵的对角元素都是1，其他元素都是0。将x替换为其QR分解，计算B的等式就变成：

$B = R^{-1} Q^T y$

这种形式的数值稳定性要高很多，尽管并不需要计算QR分解。你没有必要自己实现该计算，因为有许多不同的库可供使用。例如，Java就包含Apache Commons Math库，可以通过Maven来获得：

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-math3</artifactId>
  <version>3.2</version>
</dependency>
```

使用Apache Commons Math库的QR分解类，很容易直接得到LinearModel拟合实现中的最终等式：

```
public void fit(double[] y,double[][] x) {
  RealMatrix X = new Array2DRowRealMatrix(x);
  RealVector Y = new ArrayRealVector(y);
  B = (new QRDecomposition(X)).getSolver().solve(Y).toArray();
}
```

当然，Apache Commons Math库还包含了一个实现最小二乘法的类，因此也可以直接使用这个类：

```
public void fitOLS(double[] y,double[][] x) {
  OLSMultipleLinearRegression ols = new OLSMultipleLinearRegression();
  ols.newSampleData(y, x);
  B = ols.estimateRegressionParameters();
}
```

为了在流数据环境中使用，Apache Commons Math库还包含了一个“在线”版本的最小二乘法求解程序。该程序以它的发明者Miller打头，称为Miller Updating Regression。它的发明基于这个事实：可以用新数据更新QR分解。这使得可以将观察值流入模型，当流入足够数量的观察值（至少是模型系数数量的两倍）时，就可以生成回归系数B。要使用该程

序，只需要用模型中的系数数量来初始化模型：

```
int k = 14;
MillerUpdatingRegression rm = new MillerUpdatingRegression(k,true);
```

当新数据到来时，addObservation方法以大小为k的数组x和观察到的输出y作为输入：

```
rm.addObservation(x, y);
```

要获取当前的拟合模型，需要使用regress方法。这样就会返回类似于OLDMultiple LinearRegression类的RegressionResults类，可以用该类来获取参数估计以及你感兴趣的其他值：

```
double B[] = rm.regress().getParameterEstimates();
```

11.1.3　逻辑回归

对于不需要假定观察值服从正态分布的模型，也可以使用广义线性模型（Generalized Linear Model，GLM）来拟合。在这类模型中，都假设观察值y服从指数分布族中的分布。这样的分布包括了第9章提到过的许多著名分布。例如，泊松回归就用来对计数数据进行建模。

其中一个最流行的GLM是逻辑回归模型，它用来为伯努利分布的数据建模。具体来说，它被用来对事件发生的概率或观察值属于某个类的概率建模。逻辑回归模型也有针对多类（multiclass）建模的扩展，在该扩展模型中，模型的输出是多项分布，而不是伯努利分布。与其他的线性模型类似，概率$\hat{y}$是通过对输入x的值进行加权线性组合得来的。但是，逻辑回归并不是直接使用这个概率，而是将它转换为0和1之间的值。下面是对

LinearModel类的修改：

```
public double y(double[] x) {
  double y = 0;
  for(int i=0;i<B.length;i++) y += B[i]*x[i];
  return logit(y);
}
```

其中logit方法的实现为：

```
public static double logit(double y) {
  return 1.0/(1.0 + Math.exp(-y));
}
```

当B的值已知时，逻辑回归的实现很简单。但如果给定输入和输出的观察值，需要找到合适的B，这就要比多元线性回归模型难得多。最常见的方法依靠所谓的拟牛顿法。如果要使用这个方法，最好直接使用别人实现的现成方法。C/C＋＋和FORTRAN对于逻辑回归都有许多高质量的数值计算包，但逻辑回归的Java实现却少得很。

用逻辑回归进行回归拟合

如果无法使用上述高质量数值计算包的接口，因此不能利用原始方法进行逻辑回归，还可以使用梯度下降法来计算B的恰当值。该方法对数据进行多次迭代，每观察到一个数据都调整B，该调整量正比于各观察值的预测值$\hat{y}$与实际值y之间误差的总和。调整量由参数alpha来控制。该参数名为“学习速率”，它负责控制算法收敛到最终解的速度，其起点值通常设为0.1左右，并且必须保证大于0小于1。

```java
public LogisticRegression fit(double[][] x,double[] y) {
  //Initialize the weights
  B = new double[x[0].length];
  double lastError = Double.POSITIVE_INFINITY;
  for(int iter=0;iter<MAX_ITER;iter++) {
    double err2 = 0;
    for(int i=0;i<x.length;i++) {
      double t = y[i] - y(x[i]);
      for(int j=0;j<x[i].length;i++)
        B[j] += alpha*t*x[i][j];
      err2 += t*t;
    }
    err2 = Math.sqrt(err2);
    if(err2 - lastError < 1e-6) break;
    lastError = err2;
  }
  return this;
}
```

除非达到了迭代的最大次数，或者误差变化小到不值得继续迭代，否则就一直迭代下去。梯度下降法还有更一般的应用，即用来对人工神经网络模型的参数进行拟合，我们会在下一节对此做具体介绍。

提示　梯度下降方法可以用于所有类型的回归，而不仅仅是逻辑回归。多元回归有其他的选择，但梯度下降能够支持更复杂回归的流实现。当处理流数据时，要假设未来的数据等价于过去的数据，这样就不需要对数据进行多次迭代。

11.1.4　神经网络模型

神经网络模型也被称为人工神经网络（Artificial Neural Network，ANN），它是一类非线性模型，用来在给定输入变量集合的情况下对输出进行预测。神经网络最早的例子可以追溯到20世纪40年代早期，当时用它来解决分类问题。业界普遍认为，神经网络代表了一类非线性回归方法，并且可以应用于同样的问题域。

受人类大脑中物理神经元活动的启发，人工神经元（在神经网络术语中称为结点（unit））计算其输入的加权和的函数。该函数称为激励函数，通常选自sigmoid函数族，例如前述的逻辑函数和单位阶跃函数等。图11-3展示了神经网络中3个主流的激励函数：逻辑函数、双曲正切函数和单位阶跃函数。

结点被组织成多个层。有一个输入层，一个输出层，以及若干隐藏层。输入层用来驱动隐藏层（如果存在的话）中结点的激励函数。第一个隐藏层的结点则负责驱动下一层的激励函数，依此类推，最后的隐藏层负责驱动输出层的激励函数。通常存在一个偏置结点，该结点的作用与线性模型中的截距项相同，用来向每个神经元提供一个常数输入。

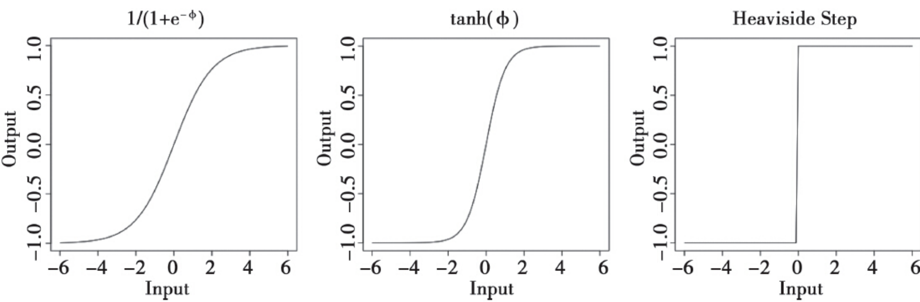


图　11-3

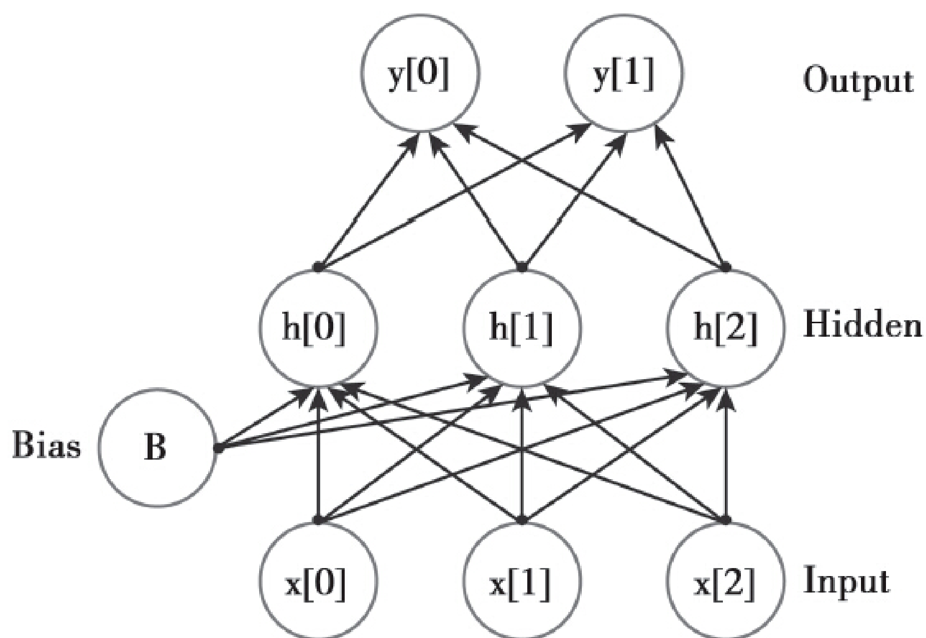


图　11-4

典型的神经网络布局如图11-4所示。这里，输入层和隐藏层都与输出层的大小相同，都只包含两个结点。通常对一层中相对于任何其他层的结点数量没有限制。因为每个结点都连接前一层的所有结点，理论上所有的结点都会学习到输入信号的各个不同部分。

11.1.4.1　多层前向传播网络的实现

前面介绍的这类神经网络称为前向传播网络。这是因为输入首先进入输入层，然后信息就通过网络一直向前传播到输出层。

为了实现该模型，首先定义一个Layer类。该类保存了该层的当前值，以及用来生成加权和的权重矩阵。权重矩阵的大小由该层的结点数量

和到该层的输入数量定义。另外，还需要有一个偏置权重数组：

```
public class Layer {

  double[]   v;
  double[][] w;
  double[]   bW = null;
  Activation fn;

  public Layer(int units,int inputs,Activation fn,boolean bias) {
    this.fn = fn;
    v = new double[units];
    w = new double[units][];
    for(int i=0;i<v.length;i++) w[i] = new double[inputs];
    if(bias)
      bW = new double[units];
  }
```

简单起见，该实现对所有结点使用相同的激励函数，这并不是必须的，但经常如此。不过，所有的函数都必须有导数，该导数会在下一节讨论的反向传播训练算法中用到。抽象类Activation为函数和其导数定义了下列方法：

```
public abstract class Activation {
  public abstract double f(double x);
  public abstract double df(double fx);
}
```

两个常用的激励函数是逻辑函数（也称为sigmoid函数）和双曲正切函数。逻辑函数的实现以一个常数k作为输入参数，它用来控制函数值从正变负的速率。通常来说，足够大的k（例如100）足以近似单位阶跃函数，该函数一般不适用于反向传播算法：

```
public static Activation logit(final double k) {
  return new Activation() {
    @Override
    public double f(double x) {
      return 1.0/(1.0 + Math.exp(-k*x));
    }

    @Override
    public double df(double fx) {
      return k*fx*(1.0-fx);
    }
  };
}
public static Activation sigmoid(double k) {
  return logit(k);
}
public static Activation logit() {
  return logit(1.0);
}
public static Activation sigmoid() {
  return logit();
}
```

双曲正切函数是激励函数的另一个热门选择。它也被包含在该实现中，作为标准的激励函数选项：

```
public static Activation tanh() {
  return new Activation() {
    @Override
    public double f(double x) {
      return Math.tanh(x);
    }

    @Override
    public double df(double fx) {
  return 1 - fx*fx;
    }
  };
}
```

注意　Activation类的实现代码中使用的导数并不是真正的f（x）的导数。这里，f（x）的导数可以用f（x）来表示，在上述代码中为1 - f（x）×f（x）。为了提高计算效率，f（x）的值被指定为fx，然后被用在“导数”函数中替代x。在神经网络领域，激励函数导数的这种表示方法

很常见。可惜的是，在相关文献中，对该导数原本是由f（x）表示的这一事实讨论得不多。因此，网络上的反向传播示例中常常存在错误。之所以使用这个版本的导数，是因为在前向传播的过程中，已经为网络中的每个结点计算出f（x）。由于必须存储这个值，才能计算每一层的误差，因此使用以f（x）表示的导数比较方便。否则，还需要存储x的原始值。

当数据输入到层中时，必须计算全部的加权和，然后对其应用激励函数。由于我们主要目的是讲解清楚计算方法，因此并不考虑性能。因此，该实现效率很低就不足为奇。它需要O（n×m）次操作：

```
public double[] feed(double[] x) {
  for(int i=0;i<v.length;i++) {
    double[] W = w[i];
    v[i] = bW != null ? bW[i] : 0.0;
    for(int j=0;j<W.length;j++) v[i] += W[j]*x[j];
    v[i] = fn.f(v[i]);
  }
  return v;
}
```

由于偏置输入始终为1，每个结点的输出值都被初始化为其偏置权重，否则就简单地置为0。一旦值得到更新，函数就返回新状态，然后该新状态就被传递给下一层（如果是最后一层，那就作为输出）。

接下来，用NeuralNetwork类将这些层组织为神经网络。该类维护一个层数组，其中至少包括一个输出层。多数网络还包含一个隐藏层，有的甚至包含多个隐藏层。没有必要定义一个单独的输入层，直接用该层中的结点就可以表示，因为其激励函数就是恒等函数。

注意　“深度学习”是有不止一个隐藏层的神经网络，也可以代表将神经网络堆叠在一起的做法。

```
public class NeuralNetwork {

  int inputUnits = 0;
ArrayList<Layer> layers = new ArrayList<Layer>();

public NeuralNetwork inputs(int inputUnits) {
  this.inputUnits = inputUnits;
  return this;
}
```

为了定义各后续层，NeuralNetwork类查看上一层（或输入的数量），来确定要使用的权重矩阵的大小：

```
public NeuralNetwork layer(int units,Activation fn,boolean bias) {
  int inputs = (layers.size() == 0) ?
    this.inputUnits : layers.get(layers.size()-1).units();
  layers.add(new Layer(units,inputs,fn,bias));
  return this;
}

public NeuralNetwork layer(int units,Activation fn) {
  return layer(units,fn,true);
}

public NeuralNetwork layer(int units) {
  return layer(units,Activation.tanh());
}
```

为了进行预测，feed方法反复对前一层的输出或输入本身计算加权和，并应用激励函数：

```
public double[] feed(double[] x) {
  for(Layer l : layers)
    x = l.feed(x);
  return x;
}
```

11.1.4.2 反向传播的学习

当然，刚构建完成的前向传播网络是没有用的，因为它还没有在任何数据上“训练”过。实际上，就目前我们实现的神经网络来说，它的输出向

量始终为0，因为网络中的所有连接都没有权重。

训练多层前向传播神经网络最流行的方法是反向传播算法（backpropagation algorithm）。该算法是广义最小二乘法的近亲，广义最小二乘法常用来拟合前面提到的回归模型。反向传播算法通过将输入值传播到整个网络之后，最小化目标输出值和观察输出值之间的平方和误差来进行训练。该误差被记录在一个误差数组中，这个数组则被加入到Layer实现中。为了效率起见，Layer实现也维护该层的总误差：

```
double[]   err;
double     E;
public double[] errors() { return err; }
```

误差数组在构造器中初始化，其大小与值数组的大小相同：

```
public Layer(int units,int inputs,Activation fn,boolean bias) {
  this.fn = fn;
  v    = new double[units];
  w = new double[units][];
  for(int i=0;i<v.length;i++) w[i] = new double[inputs];
  if(bias)
    bW = new double[units];
  err = new double[units];
}
```

在训练阶段，误差从输出层一直反向传播到输入层，这样误差数组就得以更新，算法就得名于该反向传播过程。具体来说，首先计算层中每个结点的误差，该误差为预期输出与每个结点的激励乘上激励函数导数之间的差值。对于输出层来说，该差值（被传递到s数组）是训练值与Layer类的backprop方法中的feed方法输出值的差。

```
public double[] backprop(double[] s) {
  double[] out = new double[w[0].length];
  E = 0;
  for(int i=0;i<v.length;i++) {
    err[i] = fn.df(v[i])*s[i];
    E += err[i]*err[i];
```

为了生成传播到下一个Layer的误差数组，当前Layer计算它自己误差值的加权和。如下列实现代码所示，这本质上就是前向传播的逆过程。然后该数组就被返回，这样它就可以用作下一个Layer的backprop方法的输入：

```
double[] W = w[i];
for(int j=0;j<W.length;j++) out[j] += W[j]*err[i];

  }
  return out;
}
```

当误差传播到各层时，权重就会被更新。权重的更新是根据delta法则进行的，该法则的内容是，两个结点间权重的改变正比于当前结点的误差、该结点激励函数的导数，以及输入结点的激励值的乘积。误差数组已经存储了结点误差与其导数的乘积，因此只需要再乘上前一层的激励值，在下列实现中该激励值被传递到了数组o中：

```
public double[] update(double[] o,double r) {
  for(int i=0;i<v.length;i++) {
    if(bW != null)
      bW[i] += r*err[i];
    double[] W = w[i];
    for(int j=0;j<W.length;j++)
      W[j] += r*err[i]*o[j];
  }
  return v;
}
```

另外一个传递到update方法的值r是“学习速率”，它与LogisticRegression例子中使用的学习速率相似，用来防止权重调整得过

快，有助于保持梯度下降的稳定性。对多数网络来说，r的典型值是在0.2到0.8之间。通常需要反复试验才能找到最佳值。

最后，在学习开始之前，还要进行权重的初始化。默认情况下，神经网络中所有的权重值都以0为起点，但这会导致神经网络一直徘徊在局部最优点，从而无法得到拟合数据的“最优”神经网络。通常最好是在训练前随机获得权重值，如下面代码所示。该代码也应加入Layer的实现代码中：

```
public void initialize(Random rng) {
  for(int i=0;i<v.length;i++) {
    for(int j=0;j<w[i].length;j++)
      w[i][j] = 2*rng.nextDouble() - 1;
      bW[i] = 2*rng.nextDouble() - 1;
  }
}

public void initialize() {
  initialize(new Random());
}
```

反向传播算法最终应被放在NeuralNetwork类的train方法中。然后，就可以将训练数据示例输入网络，从而得到输出。训练数据中包含一个输入数组x，以及一个输出数组y：

```
public NeuralNetwork train(double[] x,double[] y) {
  double[] out = feed(x);
```

接下来计算输出数据和输入数据的差，并沿着网络反向传播：

```
double[] s = new double[out.length];
for(int i=0;i<y.length;i++) s[i] = y[i] - out[i];
for(int i=layers.size()-1;i>=0;i--) s = layers.get(i).backprop(s);
```

最后，通过向前逐层计算，每一层的权重都得到更新：

```
    for(Layer l : layers) x = l.update(x, learningRate);
    return this;
}
```

学习异或模式

有一个神经网络学习的有趣的例子，可以用来说明隐藏层的必要性。在这个例子中，假设我们要学习的是异或（eXclusive-OR，XOR）模式。在符合该模式的训练集中，有两个输入和一个输出，只要任意一个输入是活跃的，输出就也是活跃的。只有两个输入都活跃或者都不活跃，输出才是不活跃的。

实际上，包括没有隐藏层的神经网络，许多其他的学习算法也都无法学习这种模式。这里的原因是，许多分类技术能解决问题的前提是，类之间是线性可分的，而XOR模式恰恰不是线性可分的。

向神经网络中加入隐藏层，本质上进一步细分了输入空间，从而使其学习到能由输入重建异或模式的神经网络。

下面的例子使用了两个神经网络，一个具有隐藏层，一个没有隐藏层。两者都试着学习XOR模式，它们的名称分别为nn和bad。很容易在上面的框架中定义它们：

```
NeuralNetwork nn = NeuralNetwork.build().inputs(2).layer(3).layer(1);
NeuralNetwork bad= NeuralNetwork.build().inputs(2).layer(1);

nn.initialize();
bad.initialize();
```

随后，将这两个网络在所有可能的XOR输入和输出上训练1000次，并记录误差：

```
for(int i=0;i<1000;i++) {
  double err = 0.0;
  double errBad = 0.0;
  for(Obs x : xorData) {
    nn.train(x.x,x.y);
    bad.train(x.x,x.y);
    err += nn.error();
    errBad += bad.error();
  }
  System.out.println(err+"\t"+errBad);
}
```

经过这么多次迭代之后，显然具有隐藏层的神经网络的总体误差要比没有隐藏层的神经网络小得多，后者后来始终具有较高的误差，如图11-5所示。实线展示了学习XOR模式的三层神经网络。虚线展示的是不能学习该模式的两层神经网络。

11.1.4.3　冲量反向传播

这个版本的反向传播算法在每一步更新权重时，使用最简单的方法。一个常用的性能提升方法是，向权重更新过程中引入“冲量”（momentum）的概念。

这时，层的实现不再使前面讲的基本方法来更新权重，而是用单独的矩阵来存储权重的delta值：

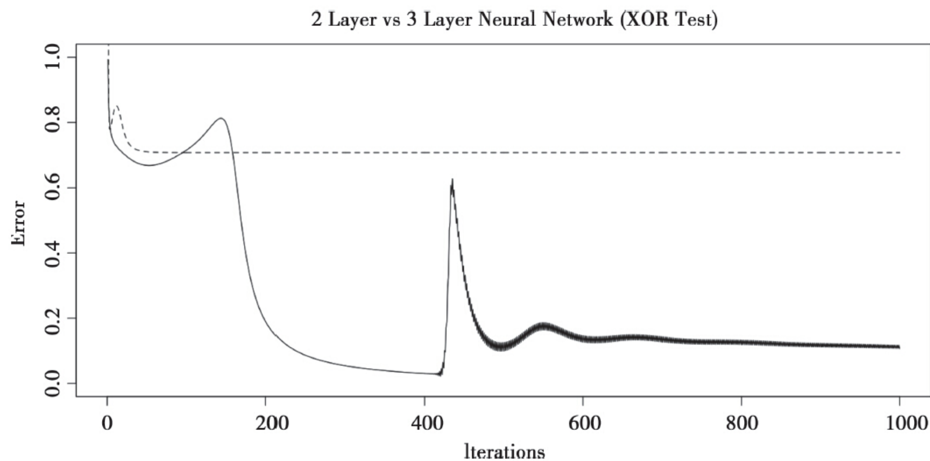


图　11-5

```
double dW[][];
double dbW[];
double m = 0.1;

public MomentumLayer(int units, int inputs,
  Activation fn, boolean bias) {
  super(units, inputs, fn, bias);
  dW = new double[units][];
  for(int i=0;i<v.length;i++) dW[i] = new double[inputs];
  if(bias)
    dbW = new double[units];
}
```

在权重的更新步骤中，改为使用该权重的新delta值与旧delta值的加权平均，其中，加权平均中用到的权值由常数m确定：

```
@Override
public double[] update(double[] o, double r) {
  for(int i=0;i<v.length;i++) {
    if(bW != null) {
      dbW[i] = (1-m)*r*err[i] + m*dbW[i];
      bW[i] += dbW[i];
    }

    double[] W = w[i];
    for(int j=0;j<W.length;j++) {
      dW[i][j] = (1-m)*r*err[i]*o[j] + m*dW[i][j];
      W[j]    += dW[i][j];
    }
  }
  return v;
}
```

这使得权重向同样的方向更新，以加快收敛速度。因为如果权重更新的方向突然改变，那么收敛速度就会放慢。在本章使用的这个小例子中，使用冲量反向传播的效果就比较一般。不过，如果所训练的神经网络在不使用冲量反向传播时收敛很慢，那么加入冲量项应该可以提高性能。

11.2　用模型预测

预测在实时数据分析中应用广泛，它或者单独预测未来值（例如支持自动容量规划），或者在后面介绍的监控或优化应用领域中，用来与观察数据相对比。最原始的预测方法实际上很简单，就是使用历史数据的平均或移动平均作为未来值的预测。对于非常简单的系统，这是足够的，但多数时候，这些原始方法无法获取底层系统的足够信息来生成好的预测。

预测这个话题很宽泛，多年以来，已经有许多著作对此作专门论述。本章涵盖了其中的一些技术，这些技术或者简单，或者有效，抑或兼而有之，因此得到广泛认可。

11.2.1　指数平滑法

指数平滑预测法得到了十分广泛的普及，这是因为它们实现起来比较容易，能够处理许多时间序列中出现的季节性（seasonality），并且当信号噪声不是很大时，通常具有较好的性能。当数据没有趋势性分量或季节性分量时，这类预测最简单的形式就是，直接用当前时间段的指数移动平均，作为下一个时间段的预测值。

当然，多数时间序列会有趋势性分量或季节性分量，甚至可能两者兼具。如果是这样的话，可以建立具有不同类型的趋势性分量和季节性分量的多种模型。其中，最著名的模型是Holt-Winteres模型。这类模型可追溯到20世纪50年代，它们假设时间序列的趋势性分量是可加的，而季节性分

量是可加或可乘的。

Holt的方法用来计算模型中归因于可加趋势性分量的那部分（本节后面会讨论季节性分量）。为了计算模型的趋势性分量，我们将预测等式分解为两个部分：水平（level）和趋势（trend）。在预测中，将x值视为朝向未来的时间步，水平起到的是简单线性回归中截距参数作用，趋势起到的是斜率参数的作用。这样，基本的预测实现就非常简单：

```
public class HoltForecast {
  double level = 0;
  double trend = 0;

  public HoltForecast(double level,double trend) {
    this.level = level;
    this.trend = trend;
  }

  public double forecast(double x) {
    return level + trend*x;
  }

}
```

要在新数据到达时更新预测值，就要将水平和趋势作为分离的指数平滑变量维护起来。平滑参数a既用于趋势性分量又用于水平分量，平滑参数b则只用于趋势性分量。每个分量都要通过预测值与观察值之间的误差来进行调整，这与神经网络中使用的梯度下降算法很像。这称为误差修正形式（error correcting form），它等价于指数平滑方法，其实现代码如下所示：

```
double a = 0.0;
double b = 0.0;
```

```
public HoltForecast(double level,double trend,double a,double b) {
  this.level = level;
  this.trend = trend;
  this.a     = a;
  this.b     = b;
}
public HoltForecast(double level,double trend) {
  this(level,trend,0.2,0.8);
}
public double error(double y) { return y - forecast(1.0); }
public double update(double y) {
  double e = error(y);
  level = level + trend + a*e;
  trend = trend + a*b*e;
  return e;
}
```

随后才将模型的季节性分量加入所谓的Holt-Winters预测中。Holt-Winters使用了两类季节性模型：可加模型和可乘模型。这两类模型都需要两个额外的参数：季节性周期s和平滑参数g。在下面的例子中，季节性周期的长度与预测的季节性分量的最初估计值被一起传递进来：

```
public class SeasonalForecast extends HoltForecast {

  double[] s;
  double   g = 0.0;
  long     t = 0;
  public SeasonalForecast(double level, double trend,
    double a, double b,double[] s,double g) {
    super(level, trend, a, b);
    this.s = s;
    this.g = g;
    this.t = 0;
  }
}
```

在该方法中，必须事先选择季节性周期，不过它的取值通常很直观。对于该方法的许多经典应用来说，根据每季度数据还是每月数据，这个周期值分别是4或12。对于实时应用来说，该周期通常是每小时或每天，因此周期值分别是24或7。

对于可加季节性预测，对未来每个时间周期的季节性估计会被加到Holt预测的输出中：

```
public class AdditiveForecast extends SeasonalForecast {

  public AdditiveForecast(double level, double trend,
      double a, double b,double[] s, double g) {
    super(level, trend, a, b, s, g);
  }

  @Override
  public double forecast(double x) {
    int h = (t + (int)Math.floor(x-1.0)) % s.length;
    return super.forecast(x) + s[h];
  }
```

初始化季节性分量时，所有分量的和应近似等于0。

对于更新步骤来说，误差仍然是观察值与预测值之间的差。季节性值的更新只取决于误差和a、g的值，因此可以独立于水平和趋势值来更新：

```
@Override
public double update(double y) {
  double e = super.update(y);
  s[t] = s[t] + g*(1-a)*e;
  t = (t + 1) % s.length;
  return e;
}
```

可乘预测要复杂一些。预测步骤要将季节性值乘上Holt预测的输出：

```
@Override
public double forecast(double x) {
  int h = (t + (int)Math.floor(x-1.0)) % s.length;
  return super.forecast(x)*s[h];
}
```

如果用该预测，使用预测值与观察值间的常规误差时，需要将水平和趋势更新的误差项除以用于预测的季节性值。为了可以使用原来的Holt预测更新，将误差项表示为前面已经计算好的商：

```
@Override
public double error(double y) {
  return super.error(y)/s[t];
}
```

在接下来的季节性更新的计算中，该误差项会被修正。跟前面将水平和趋势更新除以季节性更新的方式相同，要将季节性更新除以水平和趋势项的和：

```
@Override
public double update(double y) {
  double z = level() + trend();
  double e = super.update(y)*s[t];
  s[t] = s[t] + (1-a)*g*e/z;
  t = (t + 1) % s.length;
  return e;
}
```

可加预测的初始季节性分量应该基本上能互相抵消，而可乘预测的季节性分量的和基本上等于周期的长度。这两种模型效果都不错，但在现实数据上，可乘模型往往要比可加模型拟合得更好。

11.2.2 回归法

与指数平滑法相同的思想也可以用于回归模型。尽管回归变量的选择空间几乎是无限的，但只要采集到足够的数据来支持所选的变量，那就可以简单地从k个最近的观察值开始着手。

```
public void streamingRegressionTest() {
  int k = 14;

  double[] x = new double[k];
```

```
    double    y;
    double    t = 0.0;
    MersenneTwisterFast twist = new MersenneTwisterFast();
    MillerUpdatingRegression rm = new MillerUpdatingRegression(k,true);
    for(int i=0;i<1000;i++) {
      double base = Math.sin(t);
      y = base + 0.5*twist.nextGaussian();
      //Update the input time series
      if(i >= x.length) {
        rm.addObservation(x, y);
        if(rm.getN() > 2*x.length) {
          double B[] = rm.regress().getParameterEstimates();
          double y2 = B[0];
          for(int j=0;j<x.length;j++)
            if(!Double.isNaN(B[j+1])) y2 += B[j+1]*x[j];
          for(int j=0;i<x.length-1;j++) x[i] = x[i+1];
          x[x.length-1]=y;
        }
      } else {
        x[i] = y;
      }
      t += 2.0*Math.PI/60.0;
    }
  }
```

上面这个例子使用的是Miller方法，该方法是用来更新QR分解的技术。也可以使用前面讨论的梯度下降法，特别是当你正在使用逻辑回归而不是线性回归时。不过，Miller方法的性能也还凑合，如图11-6所示。换句话说，对于这类任务，神经网络的性能往往要更好一些，具体情况我们会在下一节介绍。在图11-6中虚线表示的是原始信号，灰点表示的是观察数据，实线表示的是预测值。

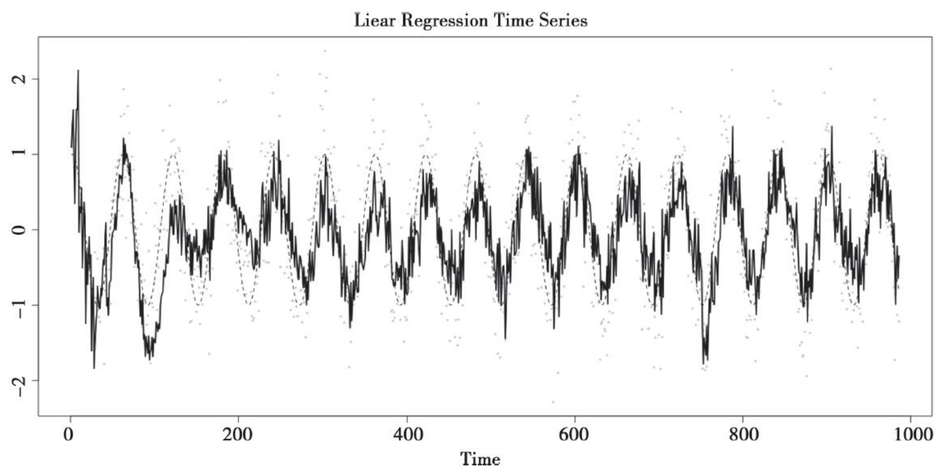


图 11-6

11.2.3 神经网络法

与回归方法类似，神经网络可以用来对时间序列数据进行预测。与回归方法一样，输入也是滞后的观察值，输出是对后面n个值的预测。例如，下列测试代码使用一个简单的神经网络，该网络用上一节中的k个观察值作为输入。输出是时间序列下一个点的预测值。

神经网络本身是用k个输入结点来初始化的，有一个隐藏层，其中包含任意选择的五个结点，还有一个输出层：

```
int k = 14;

double[] x = new double[k];
double[] y = new double[1];
double   t = 0.0;
MersenneTwisterFast twist = new MersenneTwisterFast();
NeuralNetwork       nn    = NeuralNetwork.build()
  .inputs(k)
  .layer(5)
  .layer(1)
.initialize();
```

在这个例子中，模拟数据来自正弦波，并向该信号中加入了噪声：

```
for(int i=0;i<1000;i++) {
  double base = Math.sin(t);
  y[0] = base + 0.5*twist.nextGaussian();
  //Update the input time series
  if(i >= x.length) {
    nn.train(x, y);
    System.out.println(nn.output()[0]+"\t"+y[0]+"\t"+base);
    //Move the time series over
    for(int j=0;i<x.length-1;j++) x[i] = x[i+1];
    x[x.length-1]=y[0];
  } else {
    x[i] = y[0];
  }
  t += 2.0*Math.PI/60.0;
}
```

该过程的前k步用来填充输入向量。输入向量填满之后，每当有新的输入到来，更新算法就会运行。在这个前向传播神经网络的特殊实现中，对一个观察值进行模型训练的同时，会在模型更新之前，为输出生成一个预测值。

这个例子中的神经网络是以在线的方式更新的。在该模型中，一个特殊的数据点只会使用一次，所有数据最终都会被用来改进模型。最终会生成预测值、没有噪声的基础值和作为模型输入的噪声值。图11-7同时展示了这三种值。从中可以看到，神经网络设法将原始信号完好地恢复了出来。其中，虚线表示原始信号，灰点表示实际观察到的值，实线表示神经

网络对各观察值的预测值。

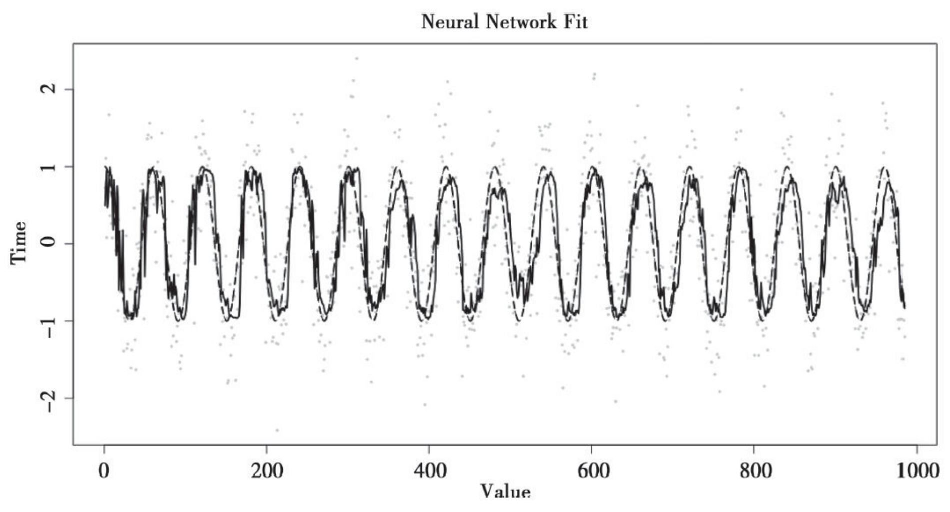


图　11-7

11.3　监控

系统监控是实时数据采集和分析的经典应用。它通常要对一段时间内的事件数量进行采集，这些数量往往会被量化为特定的频率。

例如，网站可能要采集每分钟内各类HTTP响应的数量。错误（5XX）响应频率的变化通常说明发生了某类响应，不过这种响应的发生也可能归因于极端负载高峰这样的临时事件。正确（2XX）响应频率的变化也可能说明发生了响应。对于新网站来说，“疯传”的爆炸性新闻可以有任何数量的响应。

还有经典的系统监控应用，这类应用跟踪负责运行网站的机器的运行情况。这里，跟踪的就是不同的测量值：CPU负载、风扇转速、磁盘和网络I/O。这些值的变化往往预示着外部环境的变化或硬件故障。其他更新奇的应用可能包括，医院的大规模医疗监控，或对高速公路的交通流量监控。

不论什么情况，不管什么应用领域，监控系统的目的都是很简单的：识别系统的变化，将变化通告操作人员，并使误报率最小化。它还有一个更广为人知的名称：异常检测。

11.3.1　离群点检测

最简单的异常检测是离群点检测。离群点是远离期望值的观察值，但生成数据的过程保持不变。在固定的数据集中，识别离群点通常很简单，

其方法从目视检查到拟合优度检验，不一而足。这些方法通常难以适用于在线环境。在线环境中最常用的方法是使用阈值机制来识别离群点。

这些技术的在线版本依赖于观察值和预测值之间的偏差。他们的主要差别在于，在决定某个值是真正的离群点时所用的度量。他们需要实现预测器，该预测器可以由下列接口来定义：

```
public interface Forecaster {
  public double forecast(double y);
}
```

本章讨论的任何预测机制都可以用来支持离群点检测。我们利用上一节的成果，用带有一个隐藏层的神经网络时间序列预测器来实现预测器接口：

```
public class NeuralNetworkForecaster implements Forecaster {

  NeuralNetwork nn;
  int           n;
  double[]      x;
  double[]      Y = new double[1];
  public NeuralNetworkForecaster(NeuralNetwork nn) {
    this.nn = nn;
  }

  public NeuralNetworkForecaster(int n) {
    this(NeuralNetwork.build().inputs(n).layer(n).layer(1));
    this.n = n;
    this.x = new double[n];
  }

  public NeuralNetworkForecaster() {
    this(20);
  }


  public double forecast(double y) {
    if(n > 0) {
      x[x.length - n] = y;
      y = n < x.length ? x[(x.length - n) - 1] : y;
      n--;
      return y;
    } else {
      Y[0]=y;
      y=nn.train(x, Y).output()[0];
      for(int i=0;i<x.length-1;i++)
      x[i]=x[i+1];
      x[x.length-1] = Y[0];
      return y;
    }
  }

}
```

接下来就将该预测器用作离群点检测器的基础。最简单的检测器就是简单地基于前n个观察值的误差，来跟踪与预测值相差超过若干标准差（通常为3个标准差）的值，这n个观察值存储在离群点检测器中：

```
public class ThresholdDetector {
  Forecaster f;

  int         n;
  double      sigma;
  LinkedList<Double> values = new LinkedList<Double>();
  double      s2;

  public ThresholdDetector(Forecaster f,int n,double sigma) {

    this.f = f;
    this.n      = n;
    this.sigma = sigma;
  }

 public ThresholdDetector(Forecaster f,int n) {
    this(f,n,3);
  }

 public ThresholdDetector(Forecaster f) {
    this(f,30);
  }
```

当新的观察值到达时，就通过预测器确定误差。如果误差超过sigma参数指定的标准差数量，它就会被视为离群点。这时，不会用它来更新误差的标准差，以避免导致数据偏斜。否则，就用它来更新标准差，这就表明它不是离群点：

```
public boolean observe(double y) {
  double err = y - f.forecast(y);
  double sig = Math.sqrt(s2/((double)n-1.0));
  //If this is an outlier don't include it in s2
  if(Math.abs(err)/sig > sigma)
    return true;
  //Otherwise update our standard deviation
  s2 += err*err;
  if(values.size() == n)
    s2 -= values.removeFirst();
  values.add(err*err);

  return false;
}
```

还有其他的离群点检测方法，但多数都利用上述基本框架来更新检测器。例如，许多离群点检测器不是使用标准差，而是当观察值与预测值之间的距离超过1.5或3倍的四分位范围（interquartile range）时，认定其为离群点。这个方法最初用在箱线图（boxplot）的可视化中，之后被引入到离群点检测。该方法还被扫描统计量（scan statistic）方法进一步推广，使用误差的百分比来确定过程是否处于离群点状态。

11.3.2　变化检测

在上一节讨论的离群点检测中，我们假设观察到的离群点是真正的异常，不代表生成时间序列的过程所发生的根本变化。这种变化存在的时间更长，并且通常适合使用复杂方法的可追溯方式，例如聚类或隐马尔科夫模型。变化检测的在线方法相对较少，已有的方法通常需要对数据维护至少两个预测序列，通过比较这些预测序列的误差，来确定底层的过程是否已经发生改变。

这种方法的一个例子是20世纪70年代引入金融领域的平滑异同移动平

均线（Moving Average Convergence Divergence，MACD）指标，该指标使用三个不同的指数移动平均来表征趋势的变化。这个方法原本是用来检测股票价格的趋势，但它与变化检测的现代方法非常相似。

在该方法的标准公式中，用股票的收盘价计算周期为12、26和9天的指数移动平均（EMA）。12天的EMA与26天的EMA之间的差称为MACD。9天的EMA称为信号线（signal line）。当MACD从信号线下面穿过信号线到了信号线上面时，就认为该股票转入上涨趋势。反之，就认为该股票从上涨趋势转为下跌趋势。类似的，MACD线从下跌转为上涨和从上涨转为下跌具有相同的解释，但通常该证据被视为比穿过信号线这个证据更弱。

该方法由很多变体，这些变体根据应用的情况，使用缩短或加长的EMA。不管怎样，使用该方法对趋势变化进行检测，实现起来都比较容易。该方法的主要问题是，容易错误地认定趋势变化。可以对该方法进行扩展，扩展后的方法的主要实现基本类似，但会跟踪预测值与实际值的误差趋势。这在数据中包含有已知的季节性分量时很有用，该季节性分量应该在跟踪趋势之前被删除。还有一种方法是使用Holt-Winters预测器，来跟踪模型的趋势性分量中的变化。

11.4 实时优化

如果我们正在监控某个度量的变化情况，那我们应该要能采取行动来缓解度量的不良变化，或加强度量的有利变化。否则，除了可供娱乐，对度量进行跟踪就没多少价值了。

当度量是可以由某个过程影响的结果，而该过程又可以自动控制时，更有趣的情况就出现了。例如，电子商务网站可能有两个不同的设计方案，使用两个方案可能会得到不同的收益（例如每次会话的销售量或销售金额）。测试哪个方案最好的过程被称为A/B测试，测试过程通常是启动一个新网站，跟踪其平均会话值，将其与旧网站相比较。还有一个常被大型消费网站使用的更复杂的方法，该方法将新设计的网站开放给一部分用户使用，同样对收益进行跟踪，以此进行测试。

不过，如果对每个设计，开放的用户数量是可以控制的，那么经过一段时间，就可以自动选择最佳设计，而不需要人为干预。其中一种方法就是使用所谓的单臂赌博机（multi-armed bandit）优化策略。

该策略假设玩家进入了摆满老虎机（也就是单臂赌博机）的赌场。每台老虎机都有不同的派彩率，但确定派彩（payout）率的唯一方法就是花钱在老虎机上赌博。因此，玩家追求的目标就是让自己的投资收益最大化（如果是在真实的赌场中，也可以说成是损失最小化）。直观地说，玩家刚开始会假设所有老虎机都是相同的，因此以同等的比率在上面赌博。如果某些机器的派彩率比其他的高，玩家就会逐渐把更多注意力投入到这些机器上，最终就会放弃其他机器。

再回到网站优化的例子。可以把两个不同的网站设计看作两台老虎机，“赌博”就是当有人访问网站时，将访问者指定给两个网站中的一个。这样，要解决的唯一问题就是决定访问者应该看哪个网站。

为了演示起见，我们假设访问者只能在购买和不购买之间选其一，所有商品的价格相同。这样，我们只需要对购买的概率建模，而不用直接对收益建模。每当有用户到达，就跟踪向用户开放的网站设计，假设用户已经使用下列类购买了某些商品。如下所示，用expose方法来增加接触次数，用success方法来跟踪购买量：

```
public class BernoulliThompson {
  double[] n;

 double[] x;

 public BernoulliThompson(int classes) {
   n = new double[classes];
   x = new double[classes];
 }

 public void expose(int k) {
   if(k < n.length)
     n[k] += 1.0;
 }

 public void success(int k) {
   if(k < x.length) x[k] += 1.0;
 }
```

回忆一下第9章讨论的作为伯努利试验模型的beta分布。当试验开始时，转化率是未知的，因此可能最好是将各个类的转化率指定为均匀分布，这可以用Beta（1，1）分布来建模。观察到一些访问活动之后，就可以将每个类的beta分布更新为Beta（1 + x，1 + n - x）分布。这被称为后验分布，用来为新的访问者选择网站设计。接下来根据每个网站设计的后验分布得到转化率。最后将具有最高转化率的网站设计指定给访问者，如

下所示：

```
Distribution d = new Distribution();
public int choose() {
  int    max = -1;
  double mu = Double.NEGATIVE_INFINITY;
  for(int i=0;i<x.length;i++) {
    double alpha = 1.0 + x[i];
    double beta  = 1.0 + (n[i] - x[i]);
    double x = d.nextBeta(alpha,beta);
    if(x > mu) {
     max = i;
     mu  = x;
    }
  }
  return max;
}
```

这项技术称为汤姆森抽样（Thompson sampling），它可以扩展应用于任何分布。例如，可以使用指数分布替代beta分布，从而对平均值而不是转化率进行建模。抽样过程仍然是相同的，只是将最大转化率改为从分布中抽样出来的最高派彩而已。多数情况下，这些值都很容易更新，因此网站设计的优化可以实时进行。

进行基本的估计之后，下一个逻辑步骤就是对每种网站设计应用模型，而不是使用简单的经验值。不同用户群体对各个网站设计可能具有不同的反应。这时，如果能够使用本章前面讨论的某个模型来预测平均收益或转化率，那么网站优化人员就可以为每个用户选择最佳设计，从而使网站的收益最大化。

11.5 总结

本章中的技术并非解决预测、异常检测和优化等问题的唯一方法。对于这些话题以及具体技术，都可以用整本书来论述（其实这样的书已经存在）。此外，这里提到的技术仍然是学术界和工业界中活跃的研究领域，新的改进和新的方法层出不穷。本章的目的是对这些方法进行一个简要介绍，从而为使用这些技术的相关人员提供一些基础知识，或者为他们与其他技术进行比较时提供依据。

你应该记住，即使是最简单的技术也可能有很好的结果，这一点也很重要。使用能用于大量不同问题的最简单的方法，然后再返回到最初的问题来进一步优化该方法，这样做往往效果更好。许多情况下，使用比较简单的方法就可以将最优比例从50%提高到80%，然而非常复杂方法只能将最优比例从80%提高到90%。例如，著名的Netflix大奖为能将其推荐引擎改进10%的团队提供了1百万美金的奖金。尽管后来一个团队完成了这个壮举，但他们的算法从来没有完全实现，因为实现算法的代价超过了改进算法获得的收益。